University of Alberta

YOU CAN'T ALWAYS BE RIGHT ... BUT SOMETIMES IT'D BE NICE: PREDICTING UNIX COMMAND LINES

by

Benjamin Franklin Korvemaker

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Spring 2001

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisitions et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-60446-2

Canada

University of Alberta

Library Release Form

Name of Author: Benjamin Franklin Korvemaker

Title of Thesis: You Can't Always Be Right ... But Sometimes It'd Be Nice: Predicting Unix Command Lines

Degree: Master of Science

Year this Degree Granted: 2001

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as herein before provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

Ben Korvemaker

Benjamin Franklin Korvemaker
59 Compton Road
Regina, Saskatchewan
Canada, S4S 2Y2

Date: 2000-11-29

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **You Can't Always Be Right ... But Sometimes It'd Be Nice: Predicting Unix Command Lines** submitted by Benjamin Franklin Korvemaker in partial fulfillment of the requirements for the degree of **Master of Science**.

Russell Greiner

Dr. Russell Greiner
Supervisor

Renée Elio

Dr. Renée Elio

Peter Dixon

Dr. Peter Dixon

Date: Nov. 30, 2000

If at first you don't succeed
then skydiving isn't for you.
— Anonymous

To Christie

Abstract

As every user has his own idiosyncrasies and preferences, an interface that is honed for one user may be problematic for another. To accommodate a diverse range of users, many computer applications therefore include an interface that can be customized — e.g., by adjusting parameters, or defining macros. This allows each user to have his "own" version of the interface, honed to his specific preferences. However, most such interfaces require the user to perform this customization by hand — a tedious process that requires the user to be aware of his personal preferences. We therefore explore adaptive interfaces that can autonomously determine the user's preference and adjust the interface appropriately. This thesis describes such an adaptive system — a `Unix` shell that can predict the user's next command and then use this prediction to simplify the user's future interactions. We present a relatively simple model here, and then explore a variety of techniques to improve its accuracy, including a "mixture of experts" model. In a series of experiments on real-world data, we demonstrate (1) that the simple system can correctly predict the user's next command almost 50% of the time, and can do so robustly — across a range of different users; and (2) that it is extremely difficult to further improve this result. Although we conclude that we have extracted all usable information from the dataset, improved performance may be obtained by applying the techniques herein to alternative datasets or domains.

Preface

If the reader is unfamiliar with `Unix` command lines, a brief detour to Appendix A is recommended.

Acknowledgements

I would like to thank the following for their part in this thesis:

- Julian Fogel (for his computing resources)
- Saul Greenberg (for his `Unix` command line data)
- Russell Greiner (for being my supervisor)
- Thomas Jacob (for doing the initial work with me)
- Christopher Thompson (for creating additional predictors)
- NSERC (for research funding)
- Anonymous subjects (for data collection)

Contents

1 Introduction 1

2 Scenarios 4

2.1 Document Writing Example 4

2.2 Program Writing Example 6

2.3 Current "Solutions" 7

2.3.1 Make 8

2.3.2 Aliases 8

2.3.3 Scripts 9

3 Literature Review 10

3.1 Greenberg Data 11

3.2 Programming By Example 11

3.3 Unix Command Prediction 12

3.4 Collaboration 17

3.5 Other Unix Domain Research 17

3.6 Problem with Current Machine Learning Approaches 18

4 Approach 20

4.1 Combining Experts 21

4.1.1 Correlation combiner 21

4.1.2 Democratic combiner 24

4.2 Alpha Update Rule (AUR) String Predictor 25

4.2.1 Update phase 26

4.2.2 Predict phase 30

4.3 Parsing Command Lines 31

4.3.1 Problems with Assigning Semantics 32

4.3.2 Origin of Parsing Rules 32

4.3.3 Parsing Details 32

4.3.4 How the Parsing Results Are Used 33

4.4 Other predictors 34

5 Results 35
5.1 Best predictor . . . 36
5.2 Best combiner . . . 37
5.3 Using All Predictors . . . 40
5.4 Omniscience at Work: Gods in Training . . . 42

6 Future work 44

7 Conclusion 46

Bibliography 48

Index 49

A Unix Command Line Structure 52

B AUR Predictor Example 54

C Evaluation of Command List Size 57

D Pictures of Our Implementation 61
D.1 Using the Title Bar . . . 61
D.2 Using a Separate GUI . . . 61

E Additional Predictors 64
E.1 History Predictor . . . 64
E.2 History Weighted Predictor . . . 64
E.3 Last-N Predictor . . . 65
E.4 Backwards Predictor . . . 65
E.5 Stub Predictor . . . 65
E.5.1 Stub First Predictor . . . 66
E.5.2 Stub Last Predictor . . . 66
E.6 Mode Predictor . . . 66
E.7 Cycle Predictor . . . 67
E.7.1 Entire Cycle Predictor . . . 67
E.7.2 Stub Cycle Predictor . . . 67

F Algorithms for AUR Predictor 69

G Data Collection 73

List of Tables

4.1 Predictions over time 23
4.2 Correlation combiner calculated probabilities over time 24
4.3 State of correlation table over time 24

5.1 Performance of each predictor on all users 37
5.2 Performance of each predictor on all users, ignoring training data 37
5.3 Average accuracy where the given predictor is the best predictor 38
5.4 Breakdown of best predictor to user type 38
5.5 Average accuracy where the given predictor is the best predictor, ignoring training data 38
5.6 Breakdown of best predictor to user type, ignoring training data 39
5.7 Average accuracy where the given combiner is the best combiner 39
5.8 Breakdown of best combiner to user type 40
5.9 Average accuracy where the given combiner is the best combiner, ignoring training data 40
5.10 Breakdown of best combiner to user type, ignoring training data 40
5.11 Accuracy when the correlation combiner and all predictors are used 41
5.12 Accuracy when the correlation and all predictors are used, ignoring training data . . 41
5.13 Accuracy when the the democratic combiner and all predictors are used 41
5.14 Accuracy when the the democratic combiner and all predictors are used, ignoring training data 41
5.15 Accuracy when the best combiner and all predictors are used 42
5.16 Accuracy then the best combiner and all predictors are used, ignoring training data 42
5.17 Accuracy when the best combiner and best predictors are used 42
5.18 Accuracy when the best combiner and best predictors are used, ignoring training data. 43

A.1 Breakdown of the commands Fred types in Section 2.1 53
A.2 Breakdown of the commands Wilma types in Section 2.2 53

B.1 Probability table at time 3 55
B.2 Probability table at time 4 55

B.3 Probability table at time 5 55
B.4 Probability table at time 6 55
B.5 Probability table at time 11 55
B.6 Probability table at time 23 56

C.1 Average time in seconds taken to select a command for n commands presented. . . 57
C.2 Average time in seconds taken to select a command for n commands presented when command list is inverted. 57
C.3 Average time in seconds taken to select a command for n commands presented. . . 58
C.4 Average time in seconds taken to select a command for n commands presented when command list is inverted. 58

List of Figures

2.1 Fred tries to write his thesis. 5

2.2 Example interface 6

2.3 Wilma tries to enhance `frobnicate`. 7

4.1 Example AUR Predictor Structure 26

4.2 Parsing of simple command sequence from Figure B.1 34

B.1 Sample command sequence 54

C.1 Average time in seconds taken to select a command for n commands presented. . . 58

C.2 Average time in seconds taken to select a command for n commands presented when command list is inverted. 59

C.3 Average time in seconds taken to select a command for n commands presented. . . 59

C.4 Average time in seconds taken to select a command for n commands presented when command list is inverted. 60

D.1 Title bar interface screenshot 62

D.2 Separate display screenshot 63

List of Algorithms

E.1 History Predictor 64
E.2 History Weighted Predictor 65
E.3 Backwards Predictor 65
E.4 Stub Predictor 66
E.5 Mode Predictor 67
E.6 Cycle Predictor 68
F.1 Tuples::Feed(*historyEntry*, *command*) 69
F.2 Tuples::Starve(*void*) 69
F.3 AUR-Predictor::AlphaUpdate(*command*) 70
F.4 AUR-Predictor::AddChild(*branch*, *sequence*) 70
F.5 AUR-Predictor::CreateChildren(*history*) 71
F.6 AUR-Predictor::Predict(*history*, *scale*, *incr_scale*) 72

Glossary

AUI Adaptive User Interface (Chapter 3).

AUR Alpha Update Rule (Section 4.2).

CPT conditional probability table.

GBI Graph Based Induction (Section 3.3).

GUI Graphical User Interface.

HCI Human-Computer Interaction.

IOLA Ideal Online Learning Algorithm (Section 3.3).

IPAM Incremental Probablistic Action Modeling (Section 3.3).

LMP Longest Matching Prefix (Section 3.3).

MFC Most Frequent Command (Section 3.3).

ML Machine Learning.

MRC Most Recent Command (Section 3.3).

NLP Natural Language Processing (Section 3.6).

RK Reactive Keyboard (Section 3.3).

UC Unix Consultant (Section 3.5).

AUR String Predictor Predicts command lines based on the time of day, day of week, recent command lines and recent exit codes (Section 4.2).

AUR Token Predictor AUR predictor that uses parsed command lines instead of literal command lines (Section 4.3).

Backwards Predictor Tries to build a sense of context to predict the next command line (Section E.4).

Entire Cycle Predictor Searches the recent history for a command line cycle to predict a short macro (Section E.7.1).

History Predictor Looks back at the last 100 command lines, scoring commands by frequency (Section E.1).

History Weighted Predictor Looks back at the last 100 command lines, scoring commands by weighted frequency (Section E.2).

Last N Predictor Predicts the last N unique command lines (Section E.3).

Mode Predictor Uses the last two command stubs to predict the next command line (Section E.6).

Stub Cycle Predictor Searches the recent history for a command stub cycle to predict a short macro (Section E.7.2).

Stub First Predictor Uses the command stub to predict the next command line (Section E.5.1).

Stub Last Predictor Uses the last non-option to predict the next command line (Section E.5.2).

Chapter 1

Introduction

"All mail clients suck. This one just sucks less." —Michael Elkins on mutt, circa 1995

Today there are a wide variety of interactive computer applications, ranging from web-browsers and searchers, through spreadsheets and database management systems, to editors, as well as games. As these systems become more complicated — as required to be able to accomplish more tasks, better — their interfaces necessarily also become more complex. Many of these systems have begun including tricks to help the users; e.g., if the user begins an empty file with "Dear John," Microsoft Word will suggest a "Letter" template; similarly, if the user begins a line with an asterisk (*), Word will change that character to a bullet (•) and go into its "List" environment.

Unfortunately, different users have different preferences, therefore the tricks that are appropriate for one user may be problematic for another (e.g., not all users like the fact that Word automatically formats anything starting with "http://" as a web link). Moreover, different users want to do different things with the system, as they have very different abilities, background knowledge, styles, etc. This realization — that "one size does NOT fit all" — argues for *customizable* interfaces that can provide different interfaces for different users, and hence allow each user to have an interface that is honed to his individual preferences.

Of course, many of today's application programs can be customized; e.g., most editors and shells include macro- or scripting- facilities. However, this customization process must typically be done *by the user* — this typically means that it is *not* done by the user, as this customization process:

1. requires that the user *knows how* to make this modification (e.g., knows both the names of the relevant parameters, and how to modify them),
2. requires the user to be *aware* of his specific preferences, and
3. is usually quite *tedious*.

This research project, therefore, pursues a different approach: build application systems that can *autonomously* adapt themselves to the individual users. In particular, we focus on techniques for detecting patterns in the user's interactions, and then using this information to make the interaction simpler for the user (perhaps by automatically re-setting some system parameters, or defining appropriate new macros). This thesis investigates a specific manifestation of this task: we design and implement a `Unix` command shell that predicts the user's future behavior from his previous commands, and then uses these predictions to simplify his future interactions with the shell. We have selected `ZSH` as this shell due to its clean source and powerful scripting language.

Our system is designed around three guidelines, intended to make computers easier to use:

1. We must not impede user productivity. By contrast, other "helpful" systems have a dancing mascot that pops up and stops users until they click on a button to make it go away.
2. The input to the system must be obtained unobtrusively: no questions, no interviews. While constantly monitoring the complete state of every file on the computer might help, computer performance might lag from the overhead of doing so.
3. Predictions should be made even if only a few examples exist, and those predictions should presented in a timely manner. If not, some users will turn the system off while others will simply ignore it.

With that in mind, we present a system that observes users typing command lines (and other information, such as the exit codes those commands return, the time of day, and day of week), then predicts what commands will be typed next, and provides a way for the user to use those predictions.

For example, suppose Fred is busy working on his thesis, and several times runs the command "`vi thesis.tex`" and followed by "`latex thesis.tex`". If only Fred were using our system: the computer could assign the right command ("`latex thesis.tex`") to the key F1 after Fred has typed "`vi thesis.tex`" and tell Fred this. This way, Fred would only have to press one key, rather than 17. See section Section 2.1 for a more detailed example.

Users tend to repeat the same sequences of commands, and we exploit that. Our work builds upon a system by Davison and Hirsh[7, 6, 8, 11], scaling their approach from command stubs to command lines. Using an enhancement of their algorithm, we predict command lines correctly over 45% of the time[1]. Following an alternative approach, using multiple prediction sources simultaneously, we obtain superior predictions than from a single predictor. This further enhancement can obtain an accuracy of almost 50% over a wide variety of users. We present a pair of examples in Chapter 2, illustrating the need for command line prediction, along with alternative, pre-existing solutions. Chapter 3 examines prior work, from alternative predictive interfaces to other `Unix`-related

[1] Predicting the last 5 unique command lines achieves 41% accuracy.

projects. Following, in Chapter 4, we describe the architecture of the system: the various methods of prediction, and how those prediction methods can be used together simultaneously, while algorithm performance upon the test data is discussed in Chapter 5. Future work (Chapter 6) and conclusions (Chapter 7) complete the post-mortem by arguing that the relatively simple system appears to have "gleaned" essentially all of the accuracy possible from the data collected, but that more information may be gathered from the test data we have obtained.

For the truly curious, appendices follow. Appendix A describes the `Unix` command line structure, a useful reference for those not familiar with `Unix` command lines. A detailed example of our primary predictor, the Alpha Update Rule (AUR) predictor, is presented in Appendix B. Throughout the course of research, we explored the effect of changing (a) the number and (b) the order of command line predictions on users. Appendix C is a summary of this exploration. As this research did produce a usable interface, Appendix D contains pictures of two alternate representations of that interface. We also report on a variety of other algorithms using various types of input in Appendix E, which are used by the combination methods described in Section 4.1.1 and Section 4.1.2, but are not the primary focus of research. Appendix F presents some of the grittier details of the algorithms presented in Chapter 4, and Appendix G describes our data collection processes.

Chapter 2

Scenarios

"If at first the idea is not absurd, then there is no hope for it." — Albert Einstein

With each new release, computer applications perform more tasks. Side effects of these new tasks are new configuration options, allowing greater software flexibility. However, this flexibility frequently also increases the complexity of the application (if not in performing tasks, at least in configuring the tasks). While many people may like pointing to Microsoft applications (Word, Windows, etc.) as the epitome of application complexity, the `Unix` command line suffers from similar afflictions — in fact, it may be worse. The `Unix` command line environment is a legacy system, having been around for decades, and as such, has accumulated unusual behaviors and features throughout its evolution.

For every command the user types, the computer does something. If we watch what the user types, we notice that he frequently types the same sequences of commands over and over. Given that we *can* watch what the user types, and that computers are good at recalling what has been typed before, we can estimate the probability the $t+1^{st}$ command will be χ, given the previous commands: $P(command_{t+1} = \chi \,|command_1, \ldots, command_t)$ for all possible commands χ. We can take the best one (or even the best five) and help the user by assigning those commands to otherwise unused keys, the F-keys. By doing so, we can reduce mistakes and typing time, thereby improving user productivity. Here we present examples of how different commands use different options (quickly confusing novice users) and how typographical errors may occur — and how we can easily avoid those problems. In Section 2.3, we explore what solutions are already available, and why those solutions are not used.

2.1 Document Writing Example

Fred is trying to write his thesis using LaTeX (a typesetting program whose existence spans three decades and whose use requires multiple steps[1]). Typically, the steps for using LaTeX are:

[1] By splitting each task into small steps, small changes may not require the entire task to be performed again. Instead, only relevant subtasks are reexecuted, saving time. When LaTeX was first developed in the early 1980's, this was important.

```
1   vi thesis.tex
2   latex thesis.tex
3   dvips -q thesis.dvi -t letter -o thesis.ps
4   ghostview thesis.ps
5   vi thesis.tex
6   latex thesis.tex
     LaTeX Error:
     !  Emergency stop.
     No pages of output
7   vi thesis.tex
8   latex thesis.tex
     LaTeX Error:
     !  Emergency stop.
     No pages of output
9   vi thesis.tex
10  latex thesis.tex
11  dvips -q thesis.dvi -t letter -o thesis.ps
12  ghostview thesis.ps
13  vi thesis.tex
14  latex thesis.tex
     LaTeX Error:
     !  Emergency stop.
     No pages of output
15  mail -s 'I need latex help' wilma
     My thesis keeps blowing up!
     .
16  logout
```

Figure 2.1: Fred tries to write his thesis.

1. Write document source (document ends in `.tex`). One common editor is `vi`.
2. Run LaTeX on ⟨`filename`⟩`.tex` to produce a device-independent file, creating ⟨`filename`⟩`.dvi`.
3. Run `dvips` on ⟨`filename`⟩`.dvi` to produce a PostScript file, creating ⟨`filename`⟩`.ps`.
4. View PostScript file with `ghostview`, displaying what the document will look like.

At any point (particularly when the document is not complete), the user may return to step 1. When the user is satisfied with the results at step 4, he will usually then print the document. In Figure 2.1 (`typewriter text` lines are commands typed, indented lines are relevant output messages from programs), we see Fred attempting to go through these steps. In the example, a breakdown of the parameters provided to the command can be seen in Table A.1.

At first, Fred has no problems with the usual steps (lines 1–4), but does not like the resulting document. After making some changes (line 5), LaTeX has problems with `thesis.tex` (line 6), requiring Fred to make changes to fix the problem. Once those problems are fixed (lines 7–10), LaTeX likes the document but Fred does not. Consequently, he makes more changes and it is no surprise that

There are many other good reasons for this model, but that would be a thesis in itself.

F1 [dvips peqnp] F2 [bibtex peqnp] F3 [vi peqnp.tex] F4 [latex peqnp] F5 [ls]
(/usr/lib/texmf/texmf/tex/latex/base/article.cls
Document Class: article 1996/10/31 v1.3u Standard LaTeX document class
(/usr/lib/texmf/texmf/tex/latex/base/size10.clo)) (peqnp.aux)
(/usr/lib/texmf/texmf/tex/latex/base/omscmr.fd) [1] (peqnp.aux))
Output written on peqnp.dvi (1 page, 804 bytes).
Transcript written on peqnp.log.
[134]%
"/tmp/z
This is dvipsk 5.58f Copyright 1986, 1994 Radical Eye Software
' TeX output 1999.01.18:2315' -> peqnp.ps
<tex.pro>. [1]
[135]%
"/tmp/z
[136]%
"/tmp/z
[137]%
"/tmp/z
This is TeX, Version 3.14159 (C version 6.1)
(peqnp.tex
LaTeX2e <1996/12/01> patch level 1
Babel <v3.6h> and hyphenation patterns for american, german, loaded.
(/usr/lib/texmf/texmf/tex/latex/base/article.cls
Document Class: article 1996/10/31 v1.3u Standard LaTeX document class
(/usr/lib/texmf/texmf/tex/latex/base/size10.clo)) (peqnp.aux)
(/usr/lib/texmf/texmf/tex/latex/base/omscmr.fd) [1] (peqnp.aux))
Output written on peqnp.dvi (1 page, 804 bytes).
Transcript written on peqnp.log.
[138]%
"/tmp/z

Figure 2.2: Example interface

LaTeX is unable to process `thesis.tex` again (line 14). At this point, Fred asks Wilma for help (line 15) and quits (line 17). Our system can simplify the `latex-dvips-ghostview` sequence by assigning the right command to F1 each time. This lets Fred focus on writing his thesis rather than on compiling it.

Figure 2.2 shows our interface. Here, the titlebar shows a list of predicted commands and the F-keys those commands have been mapped to.

2.2 Program Writing Example

Another common situation involving repeated multiple steps is program writing. Many computer languages require the following steps:

1. Write the program source code.
2. Compile each source code file to an intermediate format called object code.
3. Combine all object code into the destination program. This is called linking.
4. Run the program.

```
1   vi frobnicate.c
2   vi xfrobnicate.c
3   cc -c frobnicate.c
4   cc -c xfrobnicate.c -I/usr/X11R6/include
5   ld -o xfrobnicate -L/usr/X11R6/lib -lXt -lX11
6   xfrobnicate
7   vi xfrobnicate.c
8   cc -c xfrobnicate.c -I/usr/X11R6/include
9   ld -o xfrobnicate -L/usr/X11R6/lib -lXt -lX11
10  xfrobnicate
11  vi xfrobnicate.c
12  cc -c xfrobnicate.c -I/usr/X11R6/incldue
     xfrobnicate.c:2 X11/Xlib.h:  No such file or directory
     xfrobnicate.c:3 X11/Intrinsic.h:  No such file or directory
13  cc -c xfrobnicate.c -I/usr/X11R6/include
14  ld -o xfrobnicate -L/usr/X11R6/lib -lX11
     xfrobnicate.o:  In function 'main':
     xfrobnicate.o(.text+0x3b):  undefined reference to 'XtInitialize'
     collect2:  ld returned 1 exit status
15  ld -o xfrobnicate -L/usr/X11R6/lib -lXt -lX11
16  xfrobnicate
```

Figure 2.3: Wilma tries to enhance `frobnicate`.

Although these steps seem very similar to the LaTeX example above, the steps may be significantly worse. The command sequence in Figure 2.3 shows Wilma working on a program called `xfrobnicate`, adding Graphical User Interface (GUI) capability to an old program. A breakdown of the parameters provided to the commands can be seen in Table A.2.

Similar to Fred, Wilma has no problems at first (lines 1–11). However, she starts making mistakes typing out the various commands to compile `xfrobnicate`. She misspells "include" as "incldue" (line 12), and she forgets to tell the compiler to use the "Xt" library when linking `xfrobnicate` (line 14). A command shell that predicted Wilma's next command accurately would reduce typing mistakes significantly: pressing F1 or F2 is much easier than typing "`cc -c xfrobnicate.c -I/usr/X11R6/include`".

Commands are difficult to remember (and consequently type correctly). A shell that allows the right command to be executed with one key press would be useful.

2.3 Current "Solutions"

As seen above, (1) users frequently type similar (morein, the same) commands and (2) these commands can be long and complicated — which often leads to errors. We need to simplify this (1) for efficiency and (2) for reduced mistakes. Naturally, we are not the first to realize that this is somewhat inefficient, and others have presented solutions that try to eliminate errors and save time. However, the only widely deployed solutions have been limited in their acceptance. Below we re-

view systems that allow users to *manually* simplify the interface. Chapter 3 then discusses earlier attempts at *automatic* simplification.

2.3.1 Make

In the case of compiling programs, `make` has been created. A configuration file called a `makefile` contains of rules that specify what steps must be taken to get from a set of source code files, to an intermediate set of object code files, to an executable. `make` also has a number of built-in rules that help simplify `makefiles`. Although this solution has been around for decades, most undergraduate students still do not use it. For the example in Figure 2.3, a reasonable `makefile`[2] would be:

```
CFLAGS=-I/usr/X11R6/include
LDFLAGS=-L/usr/X11R6/lib -lXt -lX11
xfrobnicate: frobnicate.o xfrobnicate.o
```

"`make`" has enough built-in rules that Wilma would simple have to run "`make`" (without any arguments!). Unfortunately, many people do not know how to use "`make`", and of those that do, many are too lazy to use it for small projects.

2.3.2 Aliases

For other cases, users may take advantage of shell *alias* facilities to build their own macros. The macros may even take arguments, becoming their own mini-commands. For the example in Figure 2.1, the following alias could be created:

```
alias doc='latex $1.tex && \
    dvips -q $1.dvi -t letter -o $1.ps && \
    ghostview $1.ps'
```

This alias allows Fred to simply type "`doc thesis`" to run the three frequent steps. The double ampersands (&&) represent separation between commands and tell the shell to stop executing commands if one returns an error. This allows for a chain of commands to be run sequentially, but stopping at the first error. The backslash (\) is a continuation character which tells the shell that the alias continues on the next line. The problem with aliases is that the user must modify the resource file for their shell (a file called "`.zshrc`" for zsh users). Further compounding the situation is that the documentation for writing aliases is usually buried in an online manual (called a "man page") that many find difficult to read. Novice users have a difficulty remembering the name of the resource file and for accessing the documentation.

[2]The astute reader may notice that this `makefile` is significant shorter than most. However, sufficient information is present for `make` to properly run, as it has default rules for creating an executable from C source code.

2.3.3 Scripts

A command shell may also be used to write scripts that will be run. The same commands that are used for an alias are used for scripts. The difference between the two is that it is easier to write a script for complex tasks, whereas aliases are frequently used for simplifying single commands. Thus, the solution written as an alias could also be written as the following shell script (a file called "`ldg`"), whose contents are:

```
latex $1.tex || exit
dvips -q $1.dvi -t letter -o $1.ps || exit
ghostview $1.ps || exit
```

In this script, the "`|| exit`" tells the shell to exit only if an exit occurs. Once again, the documentation problem makes it difficult for some users to write scripts, although it alleviates the need for the user to modify their resource files.

All three solutions provided here have been around for a very long time, yet they are not used as often as they should be. Some users do not know how to apply these solutions. Others do not even realize they exist. Better education could improve the acceptance of these solutions. However, not all `Unix` users are in an educational environment. With the growing popularity of Linux, there are many home users and improved training is not an option.

Our approach is more "user-friendly." Our command line prediction functions by itself, or in conjunction with existing tools. It leverages knowledge of command lines to learn user patterns quickly, possibly before the user realizes such patterns exist. We avoid the need for users to tweak and twiddle with configuration settings and files by predicting command lines. Instead, the best five predictions (see Appendix C for why we chose five) are assigned to the F-keys. Although these mappings change after every command, they are consistent in that (1) they do change after every command, and that (2) the best prediction is assigned to F1, the second best to F2, and so on. By displaying the mappings to the user, he can then take advantage of them and simply press one key instead typing in the complete command.

This problem has been approached before, and we explore those approaches in the next chapter. However, the success of these attempts has been limited, as we shall see.

Chapter 3

Literature Review

"To steal ideas from one person is plagiarism; to steal from many is research." —Unknown

Our system may be classified as an Adaptive User Interface (AUI), observing and and adapting to the user. It contains multiple *experts* (prediction algorithms designed to exploit facets of command line patterns) that learn command line usage patterns under `Unix`-type operating systems and predict subsequent command lines. A *mixture of experts* model then uses recent expert accuracy to combine those predictions (i.e., experts that have a pattern of being wrong will have their predictions discounted, while experts that have been correct will be favored). The five most likely "next" command lines presented to the user, and the user may use the predictions or ignore them. Throughout the development, the system has been tested on a collection of real-world command lines, rather than a randomly generated set.

Langley observes that *online* learning is critical for an AUI[13]. While a tool may be able to analyze longer term data each night, its strength will come from its ability to analyze the data on-the-fly ("making them 'learning' systems, not 'learned' systems" [13, p. 368]). He goes on to say that rapid learning is needed, but CPU time is not the issue. Nor should it be, with processing power constantly increasing (the number of transistors per square inch in an integrated circuit doubles roughly every 18 months[16, "Moore's Law"], and computing power doubles roughly at the same rate). Rather, the ability to learn with a small training set — a difficult task for standard statistical methods, but not for our approach — is what will win. This is a key motivation for our work.

Users use command lines that may be difficult to type correctly, for one reason or another. Further, these commands are frequently repeated, presumably in patterns that might be detected. Given this scenario, we inspect what other researchers have done in similar or related situations.

3.1 Greenberg Data

The data we used for testing and analysis was collected by Greenberg. He collected the `Unix` traces of 168 users in 1988[9], and analyzed it for improved history facilities (although not for predictions)[10]. Despite the age of this data, it is a substantial amount: 303,628 command lines. Consequently, this data was the testing data for our work. As it did not include information we wanted to consider, we also collected data, and only at the end of research had we accumulated a similar quantity of command lines. Appendix G is a brief description of our data collection.

3.2 Programming By Example

Cypher uses programming by example with Eager, a predictive system written in Lisp for the Macintosh Hypercard environment[3, 4]. Like our system, Eager also watches user activities and tries to predict the next steps (so that it might be given the chance to perform tasks automatically). Eager records events at a high level (such as "click on the **OK** button") rather than low-level events (such as "click at screen coordinates 55,67"), and as a result, is more adaptive to window placement changes. It also uses the idea of similarity between component (button, text field, check box) labels to build up patterns (e.g., "Thursday and Friday" or "the last word of the third field"). Feedback is provided to the user by highlighting the anticipated action and uses the user's actual action to evaluate itself. The user may also use the Eager icon to have the task completed automatically. For example:

> "the user makes a list of the subjects of a stack of messages. The user selects ... and copies the subject of the first message, goes to the list, types "1." and pastes in the subject ... Then the user navigates to the second message..., selects ... and copies its subject, goes to the list, types "2.", and pastes in the second subject ... After the user navigates to the third message, the Eager icon appears ..., indicating that Eager has detected a pattern in the user's actions. Eager does not immediately begin performing actions; rather, it uses green highlighting to "anticipate" what the user is going to do next."[4, Chapter 9]

Key to Eager's design were the concepts of minimal intrusion, generalization and validation of programs without stopping user progress. These concepts were also important to this thesis work. Although our work does not involve GUI "command" prediction, we also have a system that uses user activity for feed back and that uses minimal intrusion.

Schlimmer and Hermens create a learning notepad program for pen-based computing[19]. This notepad program is designed for quickly taking notes for frequently occurring sequences, such as when recording a number of computer specifications. Since some features are only associated with

other features, providing a pick list of previously observed sequences allows user input to be improved (both in accuracy and speed). For example, after observing "Fred ran around the house," "Bill ran around the parking lot," and "Mary ran around the track," it can identify that "ran around the" should be considered a single token. This way, the user might select one of "Fred," "Bill," or "Mary" from a list, the system then fills in "ran around the," and the user could select "house," "parking lot," or "track," or they could enter a new location. By providing these predictions and allowing arbitrary input, the system is reasonably useful. Similarly, we provide a list of commands that may be selected or simply ignored. Due to differences in application domains, however, we do not build up commands a token at a time, but provide an entire command line.

Masui and Nakayama have an Emacs program that gives the user two keys: *repeat* and *predict* to improve text editing[15]. Their motivation is that users often do not recognize repetitive tasks until the tasks have been performed. The repeat key uses Emacs' built-in history mechanism to look for recent loops and re-execute them. For example, suppose a user searches for "frobnicate," changes "frobnicate" to "twiddle," and repeats these steps throughout an entire document. The predict key uses the current prefix as a match in the recent history and executes that match. Finally, they provide **no advance notification** about what these keys will do. Instead, they rely on Emacs' *undo* capability if the results are not desired. While very efficient (if the keys are not used, no overhead processing is incurred), this model is very inappropriate for our task: some `Unix` commands are highly destructive and cannot be undone, such as deleting all files with "`rm -rf`" or sending an email message. Moreover, we tell the user what commands are bound to each F-key.

3.3 Unix Command Prediction

The Reactive Keyboard (RK) by Darragh and Witten uses a tree structure for storage and calculation efficiency[5]. By assigning special meaning to a few key sequences (`ctrl-E`, `ctrl-T`, etc.), Darragh and Witten let the user type command lines and provides predictions in an auto-completion style familiar to users of modern Windows applications. However, `ctrl-E` is used to accept the completions, and `ctrl-T` is used to see alternate completions. RK is designed to improve keyboard entry for people who are unable to type quickly. However, a significant cognitive overhead is required to efficiently use the features of RK, similar to the keystroke combination requirements of Emacs. For example, when Wilma types "`cc -c xfrobnicate.c -I/usr/X11R6/include`", the next time she would only have to type "`c`" and then `ctrl-E` — unless an intervening command begins with "`c`". If that were the case, Wilma would have to repeatedly press `ctrl-T` until the correct command was displayed. However, RK is written to use `Unix` facilities available at the time, and does not compile on modern `Unix`'s such as Linux or Solaris. Contrary to Darragh and Witten's work, our desire is not to make users remember yet another obscure keystroke mapping, but rather make the interface easier to use with a minimal learning curve.

Motoda et al. have produced a GUI environment, ClipBoard, which uses Graph Based Induction (GBI) to construct predicted file operations and suggest them to the user[17, 21]. In order for ClipBoard to work, however, it does not observe command lines, but rather file I/O interactions. For example, it observes that running "`latex thesis.tex`" reads "`thesis.tex`" and writes "`thesis.dvi`". It also observes that "`xdvi thesis.dvi`" reads "`thesis.dvi`". From this, it can suggest that when "`thesis.dvi`" has been modified, "`xdvi`" should be run. The benefits of this are that no prior experience is required in order to generate scripts. However, it requires a specially modified operating system to run and lacks the light-weight, easy to plug-in approach we needed.

In their early work, Davison and Hirsh the next command stub based on the previous two *command stubs*, i.e., use $stub_{t-2}, stub_{t-1}$ to predict $stub_t$. They use *C4.5*[18], a popular algorithm for learning decision trees. After each use of C4.5, the results were then integrated into TCSH[11] (another popular Unixshell). The drawback to this approach is that it is offline learning (learning is not performed on the fly, but instead at regular intervals), and not able to adjust immediately to new patterns observed. They also mention that predicting a one-character alias does not improve user efficiency, since they use `ctrl-g` as the keystroke to insert the prediction (two keystrokes!) — a drawback our approach does not suffer from.

In an attempt to approximate online learning, they compare re-generating the decision tree every ten commands vs. re-generating daily, and a significant improvement is observed. Additionally, they use suffix-based matching (described in later papers as Longest Matching Prefix (LMP)), looking back for a matching sequence, and using the next command stub.

Later, Davison and Hirsh explore a variety of predictors[7, 6]: C4.5 (using from 1 to 4 previous commands), *Omniscient* (if a command has been seen before, it can always be predicted correctly [impossible to actually achieve]), Most Recent Command (MRC), Most Frequent Command (MFC), and LMP. All algorithms used a fixed size window (they considered $n = 100$, $n = 500$, $n = 1000$). They calculate predictions in an online fashion, using only the preceding n commands. At this point, they introduce the idea measuring microaverage (the number of commands right over the total number of commands for *all* users) and macroaverage (which is the average per person predictive accuracy, or "average of averages"). Results showed that macroaverage scores were usually (but not always) higher than microaverage scores. This indicates that users who type more commands (i.e., the "power users") are more difficult to predict. Additionally, they try not counting the first $n = 100$ commands as training data. Naturally, this allows the algorithm to improve results by skipping the warm-up period (as much as 4% for some predictors), but it also eliminates some of their test subjects from contributing results because they did not type enough commands. The also tried additional attributes, such as terminal type and machine name, but interestingly, these did not improve results.

They present a learning apprentice, `ilash` (Inductive Learning Apprentice SHell), which learns by observing the users interactions (both the commands entered, and pertinent state information from the shell[1]), rather than requiring any special training, but requires offline learning with C4.5. This is done in the same vein as programming by example (see Section 3.2). Davison and Hirsh raise concern over the time for online computation of predictions (i.e., running C4.5 after *each* command, producing significant computational overhead). As we show in the results (Chapter 5) of this thesis, this is not a concern for us.

Davison and Hirsh provide a better learning algorithm after finding C4.5 performance poor[8]. The best results from C4.5 are 38.5% macroaverage (37.2% microaverage), having trained on $\langle Command_{i-2}, Command_{i-1}\rangle \Rightarrow Command_i$ with their own set of 168,000 command stubs from 77 users. Although this was an application of a well-known Machine Learning (ML) algorithm, the overhead was high, and did not provide incremental updates. They present the description of an Ideal Online Learning Algorithm (IOLA) and their implementation of one: Incremental Probablistic Action Modeling (IPAM). The description assumes $Command_{i+1}$ is seen to follow $Command_i$. Whenever an unseen $Command_i$ is seen, they initialize the conditional probability table (CPT) for $\langle Command_{i-1}, Command_i\rangle$ from a default distribution (command stub frequency with no regard for proceeding command stubs). Updating a CPT has two steps: (1) multiply all elements $\langle Command_{i-2}, Command_{i-1}\rangle \Rightarrow \chi$, for all χ by α, (2) add $1-\alpha$ to the row $\langle Command_{i-2}, Command_{i-1}\rangle \Rightarrow Command_i$. This provides approximates a frequency-based CPT. $\alpha = 0.8$ is empirically determined to be best.

For example, assume we have the sequence of commands observed in Figure 2.1. After the third command (`dvips`) has been typed, the command distribution is as follows: `vi` $\frac{1}{3}$, `latex` $\frac{1}{3}$, `dvips` $\frac{1}{3}$. The entries:

$$\begin{aligned}
\langle \texttt{vi}, \texttt{latex}\rangle &\Rightarrow \texttt{vi}\,(33.3\%)\\
\langle \texttt{vi}, \texttt{latex}\rangle &\Rightarrow \texttt{latex}\,(33.3\%)\\
\langle \texttt{vi}, \texttt{latex}\rangle &\Rightarrow \texttt{dvips}\,(33.3\%)
\end{aligned}$$

are then added to the prediction table. That is, the next time the `vi,latex` sequence is seen, `vi`, `latex`, and `dvips` will each be predicted with probability 33.33%. After the fourth command (`ghostview`), four more entries are added to the prediction table:

$$\begin{aligned}
\langle \texttt{latex}, \texttt{dvips}\rangle &\Rightarrow \texttt{vi}\,(25\%)\\
\langle \texttt{latex}, \texttt{dvips}\rangle &\Rightarrow \texttt{latex}\,(25\%)\\
\langle \texttt{latex}, \texttt{dvips}\rangle &\Rightarrow \texttt{dvips}\,(25\%)\\
\langle \texttt{latex}, \texttt{dvips}\rangle &\Rightarrow \texttt{ghostview}\,(25\%).
\end{aligned}$$

[1]The data considered pertinent was machine name, time, date, terminal type, and current directory.

The sequence of steps continues, and after the sixth command (`latex`) is typed, the table looks like:

$$\begin{aligned}
\langle \texttt{vi}, \texttt{latex} \rangle &\Rightarrow \texttt{vi}\,(33.3\%)\\
\langle \texttt{vi}, \texttt{latex} \rangle &\Rightarrow \texttt{latex}\,(33.3\%)\\
\langle \texttt{vi}, \texttt{latex} \rangle &\Rightarrow \texttt{dvips}\,(33.3\%)\\
\langle \texttt{latex}, \texttt{dvips} \rangle &\Rightarrow \texttt{vi}\,(25\%)\\
\langle \texttt{latex}, \texttt{dvips} \rangle &\Rightarrow \texttt{latex}\,(25\%)\\
\langle \texttt{latex}, \texttt{dvips} \rangle &\Rightarrow \texttt{dvips}\,(25\%)\\
\langle \texttt{latex}, \texttt{dvips} \rangle &\Rightarrow \texttt{ghostview}\,(25\%)\\
\langle \texttt{dvips}, \texttt{ghostview} \rangle &\Rightarrow \texttt{vi}\,(40\%)\\
\langle \texttt{dvips}, \texttt{ghostview} \rangle &\Rightarrow \texttt{latex}\,(20\%)\\
\langle \texttt{dvips}, \texttt{ghostview} \rangle &\Rightarrow \texttt{dvips}\,(20\%)\\
\langle \texttt{dvips}, \texttt{ghostview} \rangle &\Rightarrow \texttt{ghostview}\,(20\%)\\
\langle \texttt{ghostview}, \texttt{vi} \rangle &\Rightarrow \texttt{vi}\,(33.3\%)\\
\langle \texttt{ghostview}, \texttt{vi} \rangle &\Rightarrow \texttt{latex}\,(33.3\%)\\
\langle \texttt{ghostview}, \texttt{vi} \rangle &\Rightarrow \texttt{dvips}\,(16.7\%)\\
\langle \texttt{ghostview}, \texttt{vi} \rangle &\Rightarrow \texttt{ghostview}\,(16.7\%).
\end{aligned}$$

Once the seventh command (`vi`) has been entered, rather than simply the current distribution of commands to the table, we update the existing entries by multiplying the $\langle \texttt{vi}, \texttt{latex} \rangle \Rightarrow \chi$ rows by $\alpha = 0.8$ and adding 0.2 to the row $\langle \texttt{vi}, \texttt{latex} \rangle \Rightarrow \texttt{vi}$. Now, the relevant $\langle \texttt{vi}, \texttt{latex} \rangle \Rightarrow \chi$ rows are:

$$\begin{aligned}
\langle \texttt{vi}, \texttt{latex} \rangle &\Rightarrow \texttt{vi}\,(46.7\%)\\
\langle \texttt{vi}, \texttt{latex} \rangle &\Rightarrow \texttt{latex}\,(26.7\%)\\
\langle \texttt{vi}, \texttt{latex} \rangle &\Rightarrow \texttt{dvips}\,(26.7\%).
\end{aligned}$$

After the eighth command (`latex`), the following rows are added (because although `vi,latex` has been seen, `latex,vi` has not):

$$\begin{aligned}
\langle \texttt{latex}, \texttt{vi} \rangle &\Rightarrow \texttt{vi}\,(37.5\%)\\
\langle \texttt{latex}, \texttt{vi} \rangle &\Rightarrow \texttt{latex}\,(37.5\%)\\
\langle \texttt{latex}, \texttt{vi} \rangle &\Rightarrow \texttt{dvips}\,(12.5\%)\\
\langle \texttt{latex}, \texttt{vi} \rangle &\Rightarrow \texttt{ghostview}\,(12.5\%).
\end{aligned}$$

The final state of the prediction table after the last command (`logout`) is:

$$\langle \texttt{vi}, \texttt{latex} \rangle \Rightarrow \texttt{vi}\,(36.7\%)$$

⟨vi, latex⟩ ⇒ latex (13.7%)

⟨vi, latex⟩ ⇒ dvips (29.7%)

⟨vi, latex⟩ ⇒ mail (20%)

⟨latex, dvips⟩ ⇒ vi (20%)

⟨latex, dvips⟩ ⇒ latex (20%)

⟨latex, dvips⟩ ⇒ dvips (20%)

⟨latex, dvips⟩ ⇒ ghostview (40%)

⟨dvips, ghostview⟩ ⇒ vi (52%)

⟨dvips, ghostview⟩ ⇒ latex (16%)

⟨dvips, ghostview⟩ ⇒ dvips (16%)

⟨dvips, ghostview⟩ ⇒ ghostview (16%)

⟨ghostview, vi⟩ ⇒ vi (26.7%)

⟨ghostview, vi⟩ ⇒ latex (46.7%)

⟨ghostview, vi⟩ ⇒ dvips (13.3%)

⟨ghostview, vi⟩ ⇒ ghostview (13.3%)

⟨latex, vi⟩ ⇒ vi (30%)

⟨latex, vi⟩ ⇒ latex (50%)

⟨latex, vi⟩ ⇒ dvips (10%)

⟨latex, vi⟩ ⇒ ghostview (10%)

⟨latex, mail⟩ ⇒ vi (31.3%)

⟨latex, mail⟩ ⇒ latex (31.3%)

⟨latex, mail⟩ ⇒ dvips (12.5%)

⟨latex, mail⟩ ⇒ ghostview (12.5%)

⟨latex, mail⟩ ⇒ mail (6.3%)

⟨latex, mail⟩ ⇒ logout (6.3%).

At this point the state of the default distribution table is:

command	frequency
vi	5
latex	5
dvips	2
ghostview	2
mail	1
logout	1

This method, IPAM, predicts with 39.9% macroaverage (38.8% microaverage), outperforming C4.5. This thesis work differs from Davison and Hirsh's work in that :

1. We extend IPAM from being a fixed depth to a dynamically learned depth: our AUR predictor.
2. We predict complete command lines, rather than just the command stubs.
3. We use other attributes, such as time of day, day of week, and command exit code.
4. Instead of a single best algorithm, we use a mixture of experts model because different algorithms are better for different users: the AUR predictor is just one of those experts.

3.4 Collaboration

Lashkari et al. present the idea of using collaboration for improving performance among agents that would normally learn from scratch for each user[14] (i.e., patterns learned from one user can be applied to new users). They go on to explain that agents that learn by "watching over the shoulder" and have a slow learning curve can benefit by finding similar peers (i.e., other users who have something in common with the user in question). For instance, suppose Fred types the command lines "`vi cw.tex`", "`latex cw.tex`" and "`dvips cw.dvi`". Later, suppose Wilma types "`vi cw.tex`" and "`latex cw.tex`". At this point, the patterns learned from Fred could be applied, and even though Wilma has *never* run "`dvips`" before, the command line "`dvips cw.dvi`" could be predicted. Although this is an interesting and potential useful approach, empirical results by both Davison and Hirsh and ourselves indicates otherwise[8, 12]. Further, although privacy between users is not an issue while testing on historical data, an interactive interface that provides predictions learned from other users can be a great security risk[2].

3.5 Other Unix Domain Research

The Unix Consultant (UC) provides help on the use of commands[2, 20]. At the time of the initial reports on UC (1989), it was still incomplete and not suitable for life in the real world. Rather, UC was a test-bed for investigating AI issues. Although the domain for UC is `Unix` commands, the UC components which model user knowledge interact with the user in English. The system is able to assume user knowledge based on what the user says, and to decide what to tell the user based on what knowledge is known or inferred. KNOME, the user modeling component of UC, tries to tailor explanations to what the user knows, building on expertise and what has been said before. Since UC is designed as a help facility, it is acceptable if it is unable to answer user questions (it is "fail-soft").

[2]More than once during our data collection, non-commands were accidently pasted into a command session. If a password was in that pasted data, it could be passed on to other users. We consider this unacceptable.

Similarly, this thesis work, while it strives to provide command line predictions, may be ignored by the user if the predictions are not appropriate (and so it is also fail-soft). However, UC does not predict command lines, while we do.

Another system, APU allows the manual construction of `Unix` commands from a Lisp-like language[1]. Although one can see the similarities (and potential integration) with UC, the application to this thesis' current work seems minimal. Rather than learning command sequences, they simply work with the ability to construct commands from another language. Its skills are more in its ability to apply known solutions or to construct new solutions by combining other solutions. While APU does construct commands lines, it does not predict command lines.

3.6 Problem with Current Machine Learning Approaches

The task at hand, predicting command lines, differs from many ML tasks in that :

1. Learning must be performed online (i.e., after each command line is observed) with minimal delay. Although offline learning algorithms (such as C4.5) can be modified for online learning, other problems exist.

2. Data points are so sparse that separation into training and testing data sets is unreasonable. Algorithms such as C4.5 will split the data into two groups, perhaps training with $\frac{2}{3}$ of the data and testing with $\frac{1}{3}$ of the data. If a user has only typed 30 commands, then only 20 commands can be used for training.

3. The range of values for command lines is great. In the Greenberg data, some users have over 1000 different command lines (one has 3160 distinct command lines), while only 10 have less than 100 different command lines. Decision tree algorithms (e.g., C4.5) perform better when variables (command lines in this case) have only a few possible values (less than 50), or when variables are continuous (i.e., numerical values).

Looking at these differences in another way, assume 100 command lines are needed for proper training, thus 150 command lines must be observed. If a user types one command line per minute, on average, and types command lines seven hours per day, five days per week, and fifty weeks per year, reasonable predictions could be expected after almost 3 hours. However, experience shows that few users type this many commands, (i.e., closer to one command line every three minutes), and it might take over **two days** for reasonable predictions.

The domain of `Unix` command lines does resemble the Natural Language Processing (NLP) domain. However, for each command, the meaning of command options (e.g., "`-q`") and arguments (e.g., "`cw.ps`") can vary greatly. Additionally, sometimes redundancies (thoroughly exploited with

NLP) exist within a command line, and sometimes they do not. As such, it is difficult to directly map NLP techniques to our task. In fact, our problem domain is that of "supervised use," where the learning task does not act autonomously, but provides a shortcut to a task for a user. Clearly, while there is overlap with other ML domains and our own problem domain, deficiencies (that we address) also remain.

As can be seen, much of the prior work touches upon portions of our work: commands are predicted, user interfaces are non-intrusive, and learning is performed by watching the user unobtrusively. However none completely solves the problems we face: we desire a complete, yet light-weight solution. In the next chapter, we see a complete description of our approach.

Chapter 4

Approach

"Two wrongs don't make a right, but three rights make a left." —Unknown

The goal of this work is simple: predict exact `Unix` command lines as accurately as possible. However, we must follow these rules:

1. Observe the user *unobtrusively* by watching the commands typed.
2. Present a new set of predictions after each command is typed.
3. Present the predictions in a manner such that they can be ignored with no ill effects.
4. No noticeable delays may occur — it should be describable as soft real-time.

The system is evaluated by the number of command lines it predicts correctly. After each command, it may predict up to five command lines. If any one of these command lines is the correct next command line (i.e., the one typed by the user), full points are awarded. If none of those command lines are correct, no points are awarded.

Although this research was motivated by Davison and Hirsh's work on predicting `Unix` command stubs, we do not predict command stubs, but rather the *complete command line*. Similar to Davison and Hirsh's work, we predict the top five command lines. Appendix C provides a full explanation of why we predict *five* command lines.

During the early work in trying to predict complete command lines, we noticed that prediction algorithms developed using data for one user would perform poorly on data from other users. After analysis, we could determine which was the best long-term predictor for a given user. However, we believed that the advantage lay in the ability to dynamically select the best combination of prediction algorithms. Individual prediction algorithms, or *experts*, could then be fine-tuned for a specific class of users, with the expectation that other experts could be fine-tuned for the remaining user classes.

Further, overlap between experts could allow for gradients between user classes (e.g., a novice user gaining experience).

For example, a novice user might only use six commands: "`vi cw.tex`", "`latex cw.tex`", "`dvips cw.tex`", "`ghostview cw.tex`", "`mail`" and "`logout`". Simply predicting the last five commands works well for such a scenario. However, suppose this user then edits a number of LaTeX files ("`vi frobnicate.tex`", "`vi foo.tex`", "`vi bar.tex`"), but still uses "`mail`" intermittently. Along with those commands, the associated "`latex`", "`dvips`" and "`ghostview`" commands are used. If the user rotates between various LaTeX files, predicting the last five commands will no longer perform adequately. However, abruptly switching to a different strategy that only predicts the "`vi-latex-dvips-ghostview`" sequence may fail to predict "`mail`" at first. Thus, a gradual transitioning between prediction strategies can perform better. Using a separate expert for each prediction strategy simplifies this.

In order to do so, we assume each expert E_i has used some aspects of the current body of information Ω (previous commands, error code, time, date, etc.), to produce its prediction for what command the user will type next:

$$P(E_i = y|\Omega)$$

where $E_i = y$ indicates that expert E_i predicts y will be the next command. Additionally, an algorithm monitors the performance of each expert, providing statistics on expert accuracy. We can use those statistics to weight the distributions when combining them, in order to pick the best predictions and leave the worst. Here we present the key components of our prediction system: the mixture of experts combination mechanism, the AUR predictor, and the parser. The assembly of these building blocks and integration of the predictors in Appendix E allows us to have predictors of varying complexity, yet maintain a clean, simple interface.

4.1 Combining Experts

During the course of research, we realized that just as no one prediction method works best for everyone, no one combination method works best for everyone. Consequently, two combiners were developed: both try to extract meta-knowledge about the predictors. One looks at combinations of predictors, and the other looks at each predictor in isolation. As with many other portions of this thesis work, simple is often better than complex.

4.1.1 Correlation combiner

The correlation combiner tries to learn when each combination of each set of predictors should be trusted, and when they should not be. For example, if there are three predictors, and every time

predictor1 and predictor2 agree and are right (i.e., prediction 1 predicts "`ls`" and predictor 2 predicts "`ls`", and the correct command is "`ls`"), then the combiner learns to favor that combination. The combiner also may learn that predictor3 is never right, and subsequently will poorly weight any predictions made by predictor3. This indirectly ranks the other predictions by predictor1 and predictor2 (but not predictor3) higher. Thus, it is theoretically possible to create a bad predictor that improves results. For example, suppose predictor 1 predicts "`ls`" and "`mail`", predictor 2 predicts "`ls`" and "`mail`", and predictor 3 predicts "`logout`" and "`mail`". Since the combination method has learned that predictor 3 makes bad predictions, "`ls`" will be weighted higher than "`mail`".

Recall, each expert E_i uses some information Ω to produce its prediction: $P(E_i = y|\Omega)$. Our goal is to combine these individual experts into a prediction $P(C = \chi|\Omega)$, where $C = \chi$ is the composite prediction. To simplify the derivation, assume there are only two experts. We can then write:

$$\begin{aligned} P(C=\chi|\Omega) &= P(C=\chi, E_1=y, E_2=z|\Omega) \\ &= \sum_{y,z} P(C=\chi|E_1=y, E_2=z, \Omega) \times P(E_1=y|E_2=z, \Omega) \times P(E_2=z|\Omega) \\ &= \sum_{y,z} P(C=\chi|E_1=y, E_2=z) \times P(E_1=y|\Omega) \times P(E_2=z|\Omega) \end{aligned}$$

where the last line uses the assumptions that the prediction will depend only on the values that the experts say (i.e., $P(C=\chi|E_1=y, E_2=z, \Omega) = P(C=\chi|E_1=y, E_2=z)$); and E_2's prediction is independent of E_1's given the background Ω — $P(E_1=y|E_2=z, \Omega) = P(E_1=y|\Omega)$. To simplify, assume that $P(C=\chi|E_1=y, E_2=z) = 0$ when $\chi \notin \{y, z\}$, and moreover, that we can lump together various $y \neq z$ cases. This means we need only estimate three additional numbers:

$$\begin{aligned} P_{=,=} &= P(C=\chi|E_1=\chi, E_2=\chi) \\ P_{=,\neq} &= p(C=\chi|E_1=\chi, E_2\neq\chi) \\ P_{\neq,=} &= P(C=\chi|E_1\neq\chi, E_2=\chi) \end{aligned}$$

(Recall $P_{\neq\neq} = P(C=\chi|E_1\neq\chi, E_2\neq\chi) = 0$ because neither E_1 nor E_2 predicts χ) . This means

$$\begin{aligned} P(C=\chi|\Omega) = \; & P_{=,=} \times P(E_1=\chi|\Omega) \times P(E_2=\chi|\Omega) \\ + \; & P_{=,\neq} \times P(E_1=\chi|\Omega) \times (1 - P(E_2=\chi|\Omega)) \\ + \; & P_{\neq,=} \times (1 - P(E_1=\chi|\Omega)) \times P(E_2=\chi|\Omega) . \end{aligned}$$

Of course, this scales to larger sets of experts. In general, if we have k experts, we need to estimate $2^k - 1$ probabilities.

After each command, the accuracies of all the predictor combinations are updated using the same alpha update method as IPAM uses. However, in the event that no predictors were right, the accuracies of the predictors are not updated (i.e., since we assume $P_{\neq,\neq} = 0$).

4.1.1.1 Correlation Combiner Example

To illustrate how the correlation combiner works, assume we have three predictors (each providing two predictions, as shown in Table 4.1), and Fred is working on another LaTeX document, his thesis. The commands he types are "`gv thesis.ps`", "`vi thesis.tex`", and "`lpr thesis.ps`". The correlation table can be seen in Table 4.3, the predictions made by each predictor can be seen in Table 4.1, and the results of combining them can be seen in Table 4.2. The sequence of events is then:

Time	Event
t1	Each probability $P_{?,?,?}$ is initialized equally (see Table 4.3), except the case of all predictors being wrong ($P_{\neq,\neq,\neq}$) is set to 0.
t2	Each predictor makes 2 predictions (Table 4.1) that are combined (Table 4.2).
t3	The user types "`gv thesis.ps`".
t4	Predictor1 and predictor2 are correct, but predictor3 has not provided a correct prediction. The correlation table (Table 4.3) is updated accordingly.
t5	Each predictor makes 2 more predictions (Table 4.1) that are combined (Table 4.2).
t6	The user types "`vi thesis.tex`".
t7	Predictor1 and predictor2 are correct, and predictor3 is incorrect again. Again, the correlation table is updated (Table 4.3).
t8	Each predictor makes 2 more predictions (Table 4.1) that are combined (Table 4.2).
t9	The user types "`lpr thesis.ps`".
t10	This time, all three predictor are correct, and the correlation table is updated (Table 4.3).

At time 1, the user has just started the shell. Predictor 1 and Predictor 2 both include the correct command ("`gv thesis.ps`") in their predictions at time 2, while Predictor 3 does not. After the first update at time 4, the combiner will now weight combinations where Predictors 1 and 2 agree on a command that Predictor 3 has not predicted. For instance, the calculation made at time 5 to

	Time		
	t2	t5	t8
Predictor 1	`gv thesis.ps` (90%) `latex thesis.tex` (10%)	`vi thesis.tex` (60%) `latex thesis.tex` (40%)	`latex thesis.tex` (80%) `lpr thesis.ps` (20%)
Predictor 2	`gv thesis.ps` (80%) `vi thesis.tex` (20%)	`gv thesis.ps` (80%) `vi thesis.tex` (20%)	`vi thesis.tex` (90%) `lpr thesis.ps` (10%)
Predictor 3	`lpr thesis.ps` (70%) `vi thesis.tex` (30%)	`lpr thesis.ps` (70%) `gv thesis.ps` (30%)	`lpr thesis.ps` (70%) `gv thesis.ps` (30%)
Actual	`gv thesis.ps`	`vi thesis.tex`	`lpr thesis.ps`

Table 4.1: Predictions over time

Time					
t2		t5		t8	
`gv thesis.ps`	(44.18%)	`vi thesis.tex`	(31.24%)	`vi thesis.tex`	(32.08%)
`lpr thesis.ps`	(31.55%)	`gv thesis.ps`	(30.16%)	`lpr thesis.ps`	(28.72%)
`vi thesis.tex`	(19.77%)	`lpr thesis.ps`	(24.56%)	`latex thesis.tex`	(28.52%)
`latex thesis.tex`	(4.48%)	`latex thesis.tex`	(14.02%)	`gv thesis.ps`	(10.66%)

Table 4.2: Correlation combiner calculated probabilities over time

Predictor Combination		Time			
		t1	t4	t7	t10
$1, 2, 3$	$P_{=,=,=}$	14.28%	11.42%	9.13%	27.30%
$1, 2, \neg 3$	$P_{=,=,\neq}$	14.28%	31.42%	45.13%	36.10%
$1, \neg 2, 3$	$P_{=,\neq,=}$	14.28%	11.42%	9.13%	7.30%
$1, \neg 2, \neg 3$	$P_{=,\neq,\neq}$	14.28%	11.42%	9.13%	7.30%
$\neg 1, 2, 3$	$P_{\neq,=,=}$	14.28%	11.42%	9.13%	7.30%
$\neg 1, 2, \neg 3$	$P_{\neq,=,\neq}$	14.28%	11.42%	9.13%	7.30%
$\neg 1, \neg 2, 3$	$P_{\neq,\neq,=}$	14.28%	11.42%	9.13%	7.30%
$\neg 1, \neg 2, \neg 3$	$P_{\neq,\neq,\neq}$	0.0%	0.0%	0.0%	0.0%

Table 4.3: State of correlation table over time

combine the three predictions is:

$$
\begin{array}{lrlll}
 & & & P(E_1 = vi) & P(E_2 = vi) & P(E_3 = vi) \\
P(C = \texttt{vi thesis.tex}) = & P_{=,=,=} = 0.1142 & & \times 0.6 & \times 0.2 & \times 0 \\
+ & P_{=,=,\neq} = 0.3142 & & \times 0.6 & \times 0.2 & \times 1 \\
+ & P_{=,\neq,=} = 0.1142 & & \times 0.6 & \times 0.8 & \times 0 \\
+ & P_{=,\neq,\neq} = 0.1142 & & \times 0.6 & \times 0.8 & \times 1 \\
+ & P_{\neq,=,=} = 0.1142 & & \times 0.4 & \times 0.2 & \times 0 \\
+ & P_{\neq,=,\neq} = 0.1142 & & \times 0.4 & \times 0.2 & \times 1 \\
+ & P_{\neq,\neq,=} = 0.1142 & & \times 0.4 & \times 0.8 & \times 0 \\
+ & P_{\neq,\neq,\neq} = 0 & & \times 0.4 & \times 0.8 & \times 1 \\
= & 31.24\% & &
\end{array}
$$

Note that the second line in this calculation represents the combination of "Predictor 1 right, Predictor 2 right, Predictor 3 wrong" ($P_{=,=,\neq}$). Since Predictors 1 and 2 are right ("`vi thesis.tex`") again while Predictor 3 is not, and the combination is further reinforced at time 7. However, for the last command (at time 9), all three combiners correctly predict that "`lpr thesis.ps`" will be the next command. At this point (time 10), the combiner reinforces the combination of all the predictors agreeing.

The state of the combination table can be seen in Table 4.3, and the predictions and their probabilities can be seen in Table 4.1.

4.1.2 Democratic combiner

The democratic combiner uses a simple proportion weighting. Suppose we use a window $n = 10$. If a predictor was right the last 8 times, its predictions are then weighted by 0.8. Thus, for all commands χ predicted by all experts E_i,

$$P(\chi|\Omega) = \sum_i P(E_i = \chi|\Omega) \times P(E_i)$$

where $P(E_i) = \frac{number\ of\ correct\ predictions}{n}$ for the last n commands.

Although the democratic combiner inherits its underlying data structures from the correlation combiner, the method of updating and combing is radically different. The approach is much more simplistic and designed to work better with large numbers of predictors by reducing the number of probabilities being estimated (such as seen in Table 4.3). Each predictor's recent performance (for variable values of recent that may differ from predictor to predictor) is used to weight the predictions from each predictor. Consequently, the correlations observed by the correlation combiner are lost. The model for the correlation combiner was that all predictors will predict something, even if confidence is not high. The great benefit of the democratic combiner is that predictors that may remain silent (unless reasonably confident) are not penalized.

4.2 Alpha Update Rule (AUR) String Predictor

The first expert is an IPAM[8] derivative. Other than for the first command ever typed by the user, this expert always has a prediction, usually based on observed patterns. In the event that no patterns match the current activity, recently seen frequent command lines are predicted. These patterns may include the command line typed, the exit code of the command line, the time of day, and/or the day of the week. If the algorithm finds that any of these attributes do not provide any information (as calculated by *information gain*), they are not used for predictions. Some users may have time-specific patterns, such as "every Monday morning, I only read email" or "from 2:00 PM to 4:00 PM, only LaTeX related commands (vi, latex, dvips, ghostview) are used." For users who do have time-dependent patterns, this knowledge can be greatly beneficial. Additionally, each `Unix` command has a numeric exit code, typically 0 indicates success while non-zero values indicate an error condition. However, some commands do not set this exit code, while the exit code from other commands is frequently ignored. For instance, if "`ls`" returns a non-zero exit code, the user is likely to ignore such information and continue with the task at hand. Alternatively, if "`netscape`" returns a non-zero exit code, this frequently indicates that the "`netscape`" process has terminated abnormally and is likely to be restarted. This model also plans for future addition of attributes, allowing the expert to determine what attributes are relevant.

After each time the user enters a command, the predictor updates its prediction mechanism (i.e., it continually learns). This allows the predictor to evolve over time to changing user activities. With the updates complete, it proceeds to make a new set of predictions, based on the day of week, the time of day, commands recently typed and the associated exit codes. The AUR string predictor has

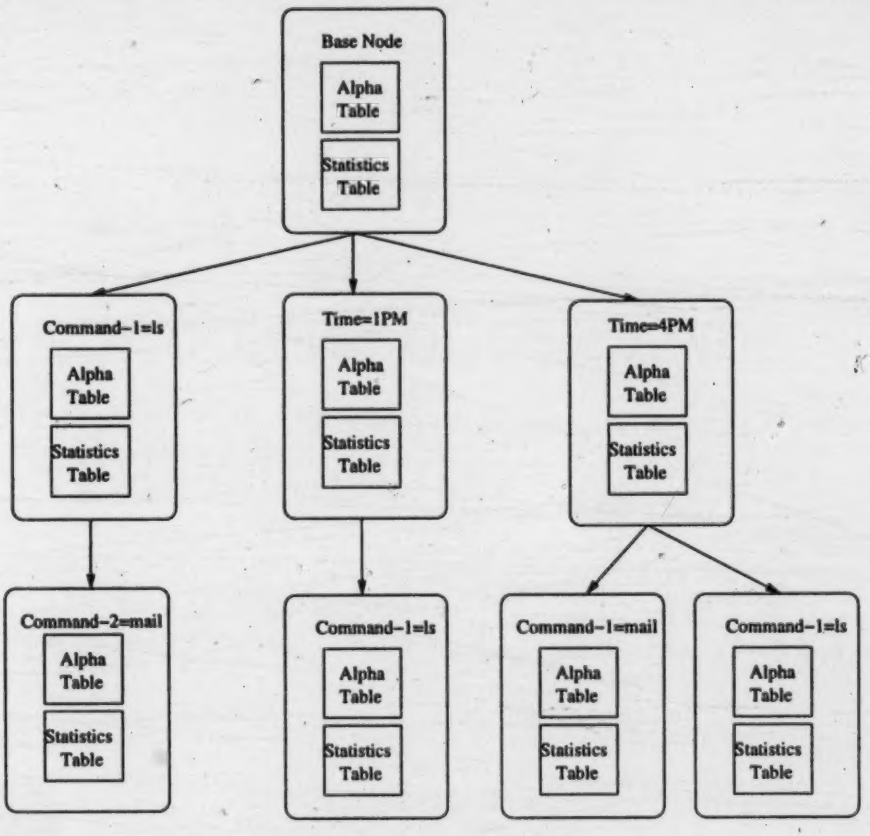


Figure 4.1: Example AUR Predictor Structure

two distinct phases: *update* and *predict*, both of which act recursively upon the tree-like structure such as the one in Figure 4.1.

4.2.1 Update phase

The update phase is further divided into four tasks: (a) update the probability table ($AlphaUpdate$); (b) update the statistics table ($TupleUpdate$); (c) update all child nodes (or branches) recursively; and (d) add child nodes ($CreateChildren$), if necessary. A description of the various processes follows.

4.2.1.1 Prediction table updates ($AlphaUpdate$)

A base table, *alphaTable*, containing the estimated probabilities of commands is maintained. This table is used for predictions when no usage patterns are observed (i.e., the values in it resemble

recent command line frequency). Additionally, until child nodes, which represent usage patterns, have been created (see Section 4.2.1.3), this is the only source of predictions. Finally, this greatly simplifies the algorithms by eliminating the need for special cases.

In order to maintain this table and provide probabilities without sufficient examples, we have adapted IPAM for table updates. After observing a command line, the probability associated with each entry in this table is decayed by multiplying it by $\alpha = 0.95$, empirically determined to be a reasonable global value. The table entry associated the command line just typed ($command_t$) then has its probability increased by $1 - \alpha = 0.05$ (if the entry does not exist, it is added with probability $= 0.05$). Should an entry's probability become sufficiently close to 0, the entry is removed from the table for efficiency (see Algorithm F.3). All entries have a combined probability of 1. This may be more succinctly viewed as

$$alphaTable \text{ at time } t,\ A^t, \text{ is a list of pairs, } \{\langle a_{command}, a_{probability}\rangle\}$$

$$A^{t+1} \leftarrow \begin{cases} \langle a_{command}, a_{probability} \times alpha\rangle \,|a \in A^t, a_{command} \neq command_t \\ \langle a_{command}, a_{probability} \times \alpha + (1-\alpha)\rangle \,|a \in A^t, a_{command} = command_t \end{cases}$$

For example, suppose the base node has an *alphaTable*:

$command_{t+1}$	probability	rank
`vi foo.tex`	49%	2
`latex foo.tex`	51%	1

That is, two commands can be predicted for the next command: "`vi foo.tex`" and "`latex foo.tex`". Further, the last command was "`dvips foo.dvi`". The *alphaTable* is first updated by multiplying the probability in each row by $\alpha = 0.95$. Next, $1 - \alpha = 0.05$ is added to the probability for the last command, "`dvips foo.dvi`". The *alphaTable* is now:

$command_{t+1}$	probability	rank
`vi foo.tex`	46.6%%	2
`latex foo.tex`	48.5%	1
`dvips foo.dvi`	5%	3

Now, suppose the next command is "`vi foo.tex`". After multiplying each entry by α and adding $1 - \alpha$ to the entry for "`vi foo.tex`", the *alphaTable* is:

$command_{t+1}$	probability	rank
`vi foo.tex`	49.2%	1
`latex foo.tex`	46%	2
`dvips foo.dvi`	4.7%	3

As this example shows, not only do the probabilities associated with each prediction change, but the rank (i.e., which is the best prediction) also changes. See Appendix B for another example of how this method works.

This approach of increasing and decreasing probabilities allows us to approximate the real frequencies without sufficient data. Additionally, it allows us to model the change in distribution as a user's activities change over time. One flaw to this method is that the first command will have an abnormally high probability. Although this could be remedied by recalculating probabilities after a small number of commands, such an event should only happen once per user's lifetime[1].

4.2.1.2 Statistical table updates ($TupleUpdate$)

Additionally, a base statistical table (used for tree growth) is maintained. This table contains *command*, *error code*, *time of day*, and *day of week*, along with a *count* of how many times each common sequence has been seen, and a *time to live* (TTL) counter. The TTL counter is decremented at every update (see Algorithm F.2) with the formula

$$D' \leftarrow \{\langle t_{command}, t_{exit\,code}, t_{day}, t_{time}, t_{count}, t_{TTL} - 1\rangle \,|\, t \in D, t_{TTL} > 0\} \qquad (4.1)$$

where $D = data$ in Algorithm F.2 and whose structure can be seen in the example below. Additionally, a list of commands, exit codes, times of day, and days of week that have been used for branching (Section 4.2.1.3) are maintained in the lists $RemovedCommands$, $RemovedExitCodes$, $RemovedTimes$, and $RemovedDays$.

If a sequence matches, the TTL counter is restored to a preset value (see Algorithm F.1), and the new table D'' is calculated by (4.2):

$$\begin{aligned}
key &\leftarrow \langle historyEntry_{command}, historyEntry_{exit\,code}, historyEntry_{time}, historyEntry_{day}\rangle \\
key_{command} &\leftarrow \bowtie \text{ if } key_{command} \in RemovedCommands \\
key_{exit\,code} &\leftarrow \bowtie \text{ if } key_{command} \in RemovedExitCodes \\
key_{time} &\leftarrow \bowtie \text{ if } key_{time} \in RemovedTimes \\
key_{day} &\leftarrow \bowtie \text{ if } key_{day} \in RemovedDays
\end{aligned}$$

$$D'' \leftarrow \{\langle t_{key}, t_{next}, t_{count} + 1, TTL\,RESET\rangle \,|\, t \in D', t_{key} = key\} \cup \{t \,|\, t \in D', t_{key} \neq key\} \qquad (4.2)$$

where D' is the result of (4.1). key undergoes changes corresponding to modifications made by the process in Section 4.2.1.3. For example, suppose a branch has been made for $command = \texttt{ls}$, the step (4.1) has been performed, and the statistics table is:

command	exit code	time of day	day of week	count	TTL
⋈	0	9 AM	MONDAY	21	19
`vi cw.tex`	0	9 AM	MONDAY	4	18
`latex cw.tex`	0	9 AM	MONDAY	4	17
`dvips cw.dvi`	0	9 AM	MONDAY	4	16
`ghostview cw.ps`	0	9 AM	MONDAY	4	15

[1] Here, we assert that patterns learned during one session will be available for use in subsequent sessions.

Additionally, suppose the last command was "ls", run at 9:45 AM on MONDAY. We then calculate $key = \langle \bowtie, 0, 9AM, MONDAY \rangle$, and update the relevant row in the statistics table by incrementing **count** and resetting the **TTL counter**:

command	exit code	time of day	day of week	count	TTL
$\bowtie$	0	9 AM	MONDAY	22	20
`vi cw.tex`	0	9 AM	MONDAY	4	1
`latex cw.tex`	0	9 AM	MONDAY	4	2
`dvips cw.dvi`	0	9 AM	MONDAY	4	3
`ghostview cw.ps`	0	9 AM	MONDAY	4	4

Suppose the next command is "`mail`", run at 9:46 AM. After the update steps, the state of the statistics table is:

command	exit code	time of day	day of week	count	TTL
$\bowtie$	0	9 AM	MONDAY	22	19
`vi cw.tex`	0	9 AM	MONDAY	4	0
`latex cw.tex`	0	9 AM	MONDAY	4	1
`dvips cw.dvi`	0	9 AM	MONDAY	4	2
`ghostview cw.ps`	0	9 AM	MONDAY	4	3
`mail`	0	9 AM	MONDAY	1	20

With the steps represented in (4.1) and (4.2) , if a common sequence is not seen for an extended period of time, it is removed from the statistics table. Suppose that "`mail`" is then run again, at 9:50AM, and the update steps are performed. At this point, the command "`vi cw.tex`" is removed from the table:

command	exit code	time of day	day of week	count	TTL
$\bowtie$	0	9 AM	MONDAY	21	18
`latex cw.tex`	0	9 AM	MONDAY	4	0
`dvips cw.dvi`	0	9 AM	MONDAY	4	1
`ghostview cw.ps`	0	9 AM	MONDAY	4	2
`mail`	0	9 AM	MONDAY	2	20

4.2.1.3 Branching for improved performance (*CreateChildren*)

After observing a command, the statistical table is checked to see if sufficient examples have been observed that might improve prediction accuracy. If such a condition exists, a child node (representing an observed pattern) is branched off the base node. The selection criteria for branching is as follows:

1. Let $C = \{c \in command_{t-1} \,|c_{count} > command_{threshold}\}$
2. $\forall c \in C$ calculate $Gain\,(S, c)$, given S, the statistical table, where

$$Gain\,(S,c) = Entropy\,(S) - \frac{|c|}{|S|} Entropy\,(c) - \frac{|\not c|}{|S|} Entropy\,(\not c)$$

and

$$Entropy\,(S) = \sum_{i=1}^{c} -p_i \log_2 p_i$$

The calculations for entropy and gain are borrowed from decision tree theory in ML. However, we are not creating decision trees.

3. Select the command c with the best gain
4. Perform similar calculations for exit code, time of day, and day of week
5. The command, exit code, time of day, or day of week with the highest gain is then selected for the branch (see Algorithm F.5).

The child created then represents the limited situations where the condition occurs. For instance, if the branch was $command_{t-1}$ = "`ls`", the probability table for that child will then only contain $command_t$ where $command_{t-1}$ was "`ls`", as in Figure 4.1. The probability table for the child is initialized from the statistics gathered about what $command_t$'s followed the branching criteria (in this case, what $command_t$'s followed "`ls`"). This bypasses the probability table skew suffered by the base node (see Algorithm F.4), and replaces IPAM's use of a default table (see Section 3.3). Aside from this difference, the child update behavior is identical to the base node, and as time goes on, will branch children of its own. Although this growth behavior resembles decision trees, it is not a decision tree: a single command may be used to update multiple nodes within the predictor, and multiple nodes may contribute to making a prediction Section 4.2.2).

The branching behavior of the predictor is then further enhanced by allowing branching on multiple attributes for a single $time_{t-1}$ (e.g., $command_{t-1}$, $exit\,code_{t-1}$). However, in order to do so easily, we enforce an arbitrary ordering: day of week, time of day, command, exit code (i.e., if branching at $command_{t-1}$ occurs, the next child branch must be $day\,of\,week_{t-2}$, $time\,of\,day_{t-2}$, $command_{t-2}$, $exit\,code_{t-2}$, or $exit\,code_{t-1}$. This eliminates the possibility of having two nodes $\langle \text{command}_{t-1} = \texttt{ls}, \text{exit code}_{t-1} = 0 \rangle$ and $\langle \text{exit code}_{t-1} = 0, \text{command}_{t-1} = \texttt{ls} \rangle$, both of which represent the same state (i.e., by requiring exit code after command, the second node cannot exist).

4.2.2 Predict phase

The prediction phase of this predictor is simpler. Remember, the predictor has a tree-like structure, beginning with a base node. The base node can make predictions, and so may the child nodes of the tree. However, a node may contribute to the predictions only if it matches the current command/exit code/time of day/day of week pattern.

Again, for simplicity's sake the base node matches all commands, and contributes to all predictions (although the extent to which it contributes may be limited, see Section 4.2.2.1). Algorithm F.6

provides the step by step details of how predictions are made by each node. However, these steps may be more simply represented as:

1. Gather predictions from children (a set of $\langle command, probabilty \rangle$).

$$\forall c \in children \,|\, P_c \leftarrow c.Predict$$

2. Combine the predictions, adding the probabilities when two or more children predict the same command.

$$\forall \chi \text{ predicted by one or more } c \in children \left| P(\chi) = \sum_{c \in children} P_c(\chi) \right.$$

3. If the child predictors have provided predictions, use those, ignoring the predictions by parent nodes.

4. Otherwise multiply each entry in the probability table by *scale* (described in Section 4.2.2.1) and use the result.

$$\forall \langle \chi, P(\chi) \rangle \in alphaTable \,|\, P\prime(\chi) = scale \times P(\chi)$$

4.2.2.1 Fine-tuning the predictions

Two parameters are available to alter the behavior of the predictor: *scale* and *scale increment*. *scale* is the scaling factor applied to the probability for a command. *scale increment* is the scaling factor applied to *scale* by a parent to a child (e.g., if $scale = 1.0$ and $scale\,increment = 2.0$, the parameters provided to the child node will be $scale = 2.0$ (i.e., $scale \times scale\,increment$) and $scale\,increment = 2.0$). Probabilities are then simply summed together, since this technique provides the best empirical results. By setting $scale\,increment > 1.0$, child nodes are favored over the parent nodes (i.e., longer chains will be favored over shorter ones). For values of $scale\,incremnt \gg 1.0$, parent nodes will only be used when child nodes cannot provide enough predictions (a further compile-time adjustment exists that avoids using parent predictions if *any* children provide predictions — we use this setting). Setting $scale\,increment < 1.0$ favors the base node, with child nodes adding a small amount of confidence to the predictions (this setting is not recommended). Setting $scale\,increment = 1.0$ is also not recommended for similar reasons.

4.3 Parsing Command Lines

From the earliest work, we believed that parsing each command line into useful components could improve prediction accuracy. In particular, the ability to predict command patterns in situations in new situations can prove immensely valuable. For example, if the sequence "`vi thesis.tex`",

"`latex paper.tex`" (see Figure B.1) is seen, and later the command line "`vi paper.tex`" is seen, the command line "`latex paper.tex`" could be predicted because of the common elements "`vi`" and "`_____.tex`". Predictions that involve the synthesis of new commands in new situations allows the predictor to "learn" seem "intelligent." Here, we present a parser whose output is used with (a copy of) the AUR prediction algorithm above to create the AUR token predictor.

4.3.1 Problems with Assigning Semantics

`Unix` command syntax has a nasty habit of changing for each command (or at least, for each set of commands written by different groups of people), making it difficult to create a semantic command line parser. For example, while many commands use "`-v`" to mean "be more verbose," some commands take it to mean "report program version." This confusion has led to some people using long command line options — but once again a split exists. Some long command options are preceded by one dash, while others are preceded by two dashes. Consequently, we do not try to learn any semantics about command line options — they are simply tokens that exist. This is not to say that this issue lies unaddressed: dozens of hard-working people maintain ZSH configuration files that properly reflect the command line options for each command (or set of commands). In time, this knowledge may be applicable to command line prediction.

4.3.2 Origin of Parsing Rules

Our command line parsing is based on personal use of `Unix` command lines, and in particular, the use of CSH history-expansion capabilities (e.g., "`!!`" to repeat the last command, or "`vi $!`" to use "`vi`" with the last argument of the last command). The parsing rules are hand-coded and ordered from simple (and expected to be most likely) to complex (both harder to identify and less likely to occur). After a match for a command line component is made, processing goes on to the next component (i.e., the first match is assumed to be the best match — in line with our "simple works well" experience). The search also looks at the most recent command lines first and only goes back a few command lines (we only go back 5 command lines for performance reasons). Although highly complex, or distant, command line patterns are not identified, we do not consider this to be a substantial issue.

4.3.3 Parsing Details

If the current command line occurs at time t, the following sequence of steps is followed for trying to find a match. In (1), all recent commands are checked. In (2), each $command_i$ $(1 \leq i \leq 5)$ is progressively checked. In all cases, as soon as a match is found, further processing stops. The steps are ordered to be progressively more complex and more processing-intensive.

1. If $command_t \in (command_{t-1}, \ldots, command_{t-8})$, select the most recent one (i.e., if $command_t = command_{t-1}$ and $command_t = command_{t-2}$, $\langle CMD - 1 \rangle$ would represent $command_{t-1}$).

2. For each whitespace-separated command line component,

   (a) See if it matches any command line component from $command_{t-1}$ (e.g., if $command_t$ ="`a.out`" and $command_{t-1}$ ="`cc a.c -o a.out`", $\langle CMD - 1, ARG\#3 \rangle$ would represent it).

   (b) Extract all combinations of removing command paths and filename extensions from $command_{t-1}$, and see if any of those match (e.g., if $command_t$ ="`foo`" and $command_{t-1}$ ="`cp /tmp/foo .`", $\langle CMD - 1, ARG\#2, \langle -PREFIX\,(1) \rangle \rangle$ would represent it).

   (c) See if removing command paths and filename extensions from $command_t$ provides a match to the combinations found in (b) (e.g., if $command_t$ = "`foo.bar`" and $command_{t-1}$ = "`cp /tmp/foo .`", $\langle CMD - 1, ARG\#2, \langle -PREFIX\,(1) +' .bar' \rangle \rangle$ would represent it).

   (d) See if adding characters to the beginning and end of command components from $command_t$ provides a match to the combinations found in (b) (e.g., if $command_t$ = "`bigfoot`" and $command_{t-1}$ ="`foo`", $\langle 'big' + CMD - 1 +' t' \rangle$ would represent it).

The example in Figure 4.2 shows the results of parsing the command sequence in Figure B.1. As can be seen in at time 16–21, tight loops can be reduced to "repeat the command that happened three commands ago," while the commands at time 0–3 can only be reduced to "use the second part of the command and prepend/append these characters."

4.3.4 How the Parsing Results Are Used

The parser provides us with a rendition of the command line based upon prior command lines, and can be used in any algorithm that is capable of using discrete command line descriptions[2]. For this thesis work, the parsed command line is used in a copy of the AUR predictor (see Section 4.2). Since it can predict a set of parsed command lines, these predictions can be reconstructed into actual command lines. For instance, if the prediction for time 24 was $\langle CMD - 3 \rangle$, the reconstructed command line is then "`./thesis`". The true benefit comes later on, when a pair of unseen command lines are seen: "`vi frobnicate.tex`" and "`latex frobnicate.tex`". The first prediction might be the parsed command line at time 1 ("`latex ...` "), and the second at time 2 ("`dvips ...`"). After reconstruction, the predictions are "`latex frobnicate.tex`" and "`dvips frobnicate.dvi -o frobnicate.ps`", allowing us to predict command lines never seen before.

[2] At this point, such command line descriptions are (a) the literal command line, (b) command stub, and (c) the parsed command line.

time	
0	`vi thesis.tex`
1	`latex` $\langle CMD-1, ARG\#2 \rangle$
2	`dvips` $\langle CMD-1, ARG\#2, \langle -SUFFIX\,(1) +' .dvi' \rangle \rangle$ `-o` $\langle CMD-1, ARG\#2, \langle -SUFFIX\,(1) +' .ps' \rangle \rangle$
3	`gv` $\langle CMD-1, ARG\#4 \rangle$
4	$\langle CMD-4 \rangle$
5	`lpr` $\langle CMD-2, ARG\#2 \rangle$
6	$\langle CMD-2 \rangle$
7	`latex` $\langle CMD-1, ARG\#2 \rangle$
8	`dvips` $\langle CMD-1, ARG\#2, \langle -SUFFIX\,(1) +' .dvi' \rangle \rangle$ `-o` $\langle CMD-1, ARG\#2, \langle -SUFFIX\,(1) +' .ps' \rangle \rangle$
9	`gv` $\langle CMD-1, ARG\#4 \rangle$
10	$\langle CMD-5 \rangle$
11	`rm` $\langle CMD-1, ARG\#2 \rangle$
12	`cd src`
13	`vi` $\langle CMD-2, ARG\#2, \langle -SUFFIX\,(1) +' .c' \rangle \rangle$
14	`cc` $\langle CMD-1, ARG\#2 \rangle$ `-o` $\langle CMD-1, ARG\#2, \langle -SUFFIX\,(1) \rangle \rangle$
15	$\langle \langle './' + \rangle, CMD-1, ARG\#4 \rangle$
16	$\langle CMD-3 \rangle$
17	$\langle CMD-3 \rangle$
18	$\langle CMD-3 \rangle$
19	$\langle CMD-3 \rangle$
20	$\langle CMD-3 \rangle$
21	$\langle CMD-3 \rangle$
22	`rm` $\langle CMD-1, ARG\#1, \langle -PREFIX\,(1) \rangle \rangle$
23	`cd ..`

Figure 4.2: Parsing of simple command sequence from Figure B.1

4.4 Other predictors

In addition to the predictors created for this thesis work, other prediction algorithms have been implemented[3]. The *history predictor* simply measures command frequency over the last 100 commands. The *weighted history predictor* is similar, but weights more recent commands higher. The *stub first predictor* uses the command stub to predict command lines, while the *stub last predictor* uses the last argument (with some limitations) on the command line to predict the next command. The *last N predictor* predicts the last 5 unique command lines. The *backwards predictor* and the *mode predictor* both attempt to identify some sense of mode or context and use that for predictions. All of these predictors follow the "simple is better" philosophy we observed during early research, relying on meta-knowledge in the combining algorithms (see Section 4.1) to identify which predictors are performing reliably and which predictors should be ignored. The details of these algorithms are documented in Appendix E.

[3]This work was performed by Christopher Thompson May–August, 2000.

Chapter 5

Results

"A program is a device used to convert data into error messages." —Unknown

The primary goal of this thesis is to produce a `Unix` shell that observes user activity and predicts command lines accurately. In order to reasonably evaluate the prediction algorithms, we used historical data, rather than interactive evaluation. This has the added benefit that future work may be benchmarked against our results. This chapter presents those results, exploring why the best predictor is the AUR string predictor (Section 4.2), and why the worst is the mode predictor (Section E.6). We also analyze why the correlation combiner (Section 4.1.1) performs poorly relative to the democratic combinerSection 4.1.2). Finally, we show how accurate we can get *if* the optimal combination of predictors and combiners could be known *a priori*.

For analysis, we have used the data collected by Greenberg (see Section 3.1). The users in this data are divided as follows:

- "Non-programmers," 25 users with minimal knowledge with `Unix`. They were office staff from computer science and faculty from environmental design. These people knew the bare minimum of `Unix` commands to get their job done.
- "Novice programmers," 55 users with little to no experience with `Unix`. These were junior undergraduate students from computer science, just learning how to use the system.
- "Experienced programmers," 36 users with some experience with `Unix`. These were primarily senior undergraduate students from computer science.
- "Computer scientists," 53 users with significant experience with `Unix`. These users represented faculty, researchers, and graduate students from computer science.

As one might imagine, the number and variety of commands differ from each group, and Greenberg has performed a detailed analysis of their history usage[10].

In the following sections, we present an analysis of our results. We determine what individual predictor is the globally best predictor, as well as what individual predictors are best for each user. Similarly, we determine which combiner is best globally and for each user. Finally, we present the result of using all predictors and each combiner, illustrating full "mixture-of-experts" approach.

5.1 Best predictor

First we consider using a single best predictor. Early research showed that although altering α values does affect the AUR algorithm performance, the difference between a globally optimal value of α and an optimal value for each user is very small. Rather, we have chosen the empirically determined optimal value for $\alpha = 0.95$ for the string and token predictors (both use the AUR prediction algorithm). Additionally, we consider ignoring the first 100 commands for each user as training, after we observed that approximately the first 100 commands are used for the AUR algorithm to "warm up."

We consider a number of cases when judging how effective our predictions are. For instance, we can observe simply the globally best predictor, as is illustrated by Table 5.1. Here, we see that the overall best predictor is the string predictor, followed closely by the history weighted predictor (Section E.2). Results do not change significantly if we ignore the first 100 commands for each user as training data, as seen in Table 5.2. Moreover, the string predictor is the best predictor for 121 (72%) of the users, getting 95163 commands (44.76% microaverage, 46.16% macroaverage) right (see Table 5.3). Thus, the best predictor is the one presented in Section 4.2.

The similarity between the two predictors may be attributed to the fact that in a degenerate case, where no noticeable patterns can be detected, the AUR string predictor follows a model very similar to the history weighted predictor. Therefore, as long as some patterns can be detected, the AUR string predictor must perform better than the history weighted predictor. The history weighted predictor performs reasonably well because the primary logic behind it is that users repeat themselves, which is the premise behind our work.

Potentially more interesting is that the last-N predictor (Section E.3) achieves almost 66% microaverage accuracy for 7 users (3 are non-programmers, 4 are novice users), and that the AUR token predictor obtains over 61% microaverage accuracy over 4 users. Although the history weighted predictor is the second best predictor, this is primarily due to it being more effective for the Experienced and Computer Scientist groups, where the last-N predictor fails miserably (see Table 5.4). It seems that while the token predictor is not effective for many users, *when* it is effective, it is very effective.

Why is the mode predictor such a bad predictor? This evaluation is unfair towards the mode predictor (Section E.6) because it does not provide predictions when its confidence is low, unlike the

Predictor	Microaverage	Macroaverage	Commands right
AUR String Predictor	43.03%	45.47%	130,651
AUR Token Predictor	38.08%	41.09%	115,612
History Weighted Predictor	41.91%	44.00%	127,250
History Predictor	39.60%	41.23%	120,224
Stub Last Predictor	31.12%	30.10%	94,503
Stub First Predictor	32.63%	32.25%	99,071
Last N Predictor	38.39%	41.64%	116,577
Backwards Predictor	24.86%	22.52%	75,480
Mode Predictor	16.00%	16.15%	48,574

Table 5.1: Performance of each predictor on all users

Predictor	Microaverage	Macroaverage	Commands right
AUR String Predictor	43.12%	46.15%	123,672
AUR Token Predictor	38.12%	41.68%	109,331
History Weighted Predictor	41.94%	44.56%	120,285
History Predictor	39.59%	41.61%	113,535
Stub Last Predictor	31.67%	31.14%	90,830
Stub First Predictor	32.85%	32.82%	94,210
Last N Predictor	38.36%	42.09%	110,037
Backwards Predictor	25.22%	22.82%	72,350
Mode Predictor	15.26%	16.85%	46,628

Table 5.2: Performance of each predictor on all users, ignoring training data

other predictors. This is why it is the worst predictor in our analysis. However, if we simply look at the times the mode predictor made a prediction, its performance is still lacking. It predicted the right command 48,574 out of a total 159,516 times. This gives a microaverage accuracy of 30.45% and a macroaverage accuracy of 30.12%. Further analysis of the algorithm reveals that it closely resembles IPAM, as it uses

$$\begin{aligned} \langle CommandStub_{t-1}, CommandStub_{t-2} \rangle &\Rightarrow Command_t \\ &\cup \\ \langle CommandStub_{t-2}, CommandStub_{t-1} \rangle &\Rightarrow Command_t\,. \end{aligned}$$

However, the "mode identification" seems to provide less information than the sequence does. Simply, although the inspiration behind the mode predictor is good, the performance is not.

5.2 Best combiner

Although the correlation combiner works reasonably well for the Greenberg dataset, for users where it is the better combiner (given the optimal combination of predictors, determined on a user-by-user basis), the total number of commands correctly predicted is 1,792. However, for users where the democratic combiner is best, the total number of commands correctly predicted is 141,731. In fact,

Best Predictor	Accuracy		Number of Users	Commands Right	Total Commands
	Microaverage	Macroaverage			
AUR String Predictor	44.76%	46.16%	121	95,163	212,618
AUR Token Predictor	61.23%	53.32%	4	2,094	3,420
History Weighted Predictor	37.46%	39.85%	28	25,661	68,504
History Predictor	43.81%	47.41%	3	425	970
Stub Last Predictor	52.12%	52.12%	1	1,968	3,776
Stub First Predictor	34.40%	36.22%	3	2,828	8,222
Last N Predictor	65.96%	62.61%	7	2,387	3,619
Backwards Predictor	32.61%	32.61%	1	815	2,499
Mode Predictor	0%	0%	0	0	0

Table 5.3: Average accuracy where the given predictor is the best predictor

Best Predictor	Non-Programmers	Novice Programmers	Experienced Programmers	Computer Scientists
AUR String Predictor	18	45	27	31
AUR Token Predictor	1	2	0	1
History Weighted Predictor	2	4	7	15
History Predictor	1	0	1	1
Stub Last Predictor	0	0	1	0
Stub First Predictor	0	0	0	3
Last N Predictor	3	4	0	0
Backwards Predictor	0	0	0	1
Mode Predictor	0	0	0	0

Table 5.4: Breakdown of best predictor to user type

Best Predictor	Accuracy		Number of Users	Commands Right	Total Commands
	Microaverage	Macroaverage			
AUR String Predictor	45.36%	47.75%	117	89,437	197,178
AUR Token Predictor	61.39%	51.08%	5	1,992	3,245
History Weighted Predictor	32.23%	42.37%	30	25,106	67,436
History Predictor	34.14%	35.31%	3	225	659
Stub Last Predictor	31.01%	31.01%	1	120	387
Stub First Predictor	39.63%	39.11%	5	4,994	12,601
Last N Predictor	67.21%	61.60%	5	1,523	2,266
Backwards Predictor	33.44%	33.29%	2	1,022	3,056
Mode Predictor	0%	0%	0	0	0

Table 5.5: Average accuracy where the given predictor is the best predictor, ignoring training data

Best Predictor	Non-Programmers	Novice Programmers	Experienced Programmers	Computer Scientists
AUR String Predictor	16	47	25	29
AUR Token Predictor	1	2	0	2
History Weighted Predictor	3	3	8	16
History Predictor	2	0	1	1
Stub Last Predictor	1	0	0	0
Stub First Predictor	0	0	1	4
Last N Predictor	2	3	0	0
Backwards Predictor	0	0	1	1
Mode Predictor	0	0	0	0

Table 5.6: Breakdown of best predictor to user type, ignoring training data

Best Combiner	Accuracy		Number of Users	Commands Right	Total Commands
	Microaverage	Macroaverage			
Correlation	44.80%	45.30%	9	1792	4000
Democratic	47.30%	46.14%	159	141731	299628

Table 5.7: Average accuracy where the given combiner is the best combiner

for almost 95% of the users (159 of the total 168), the best combiner is the democratic combiner. If the number of commands correct is counted instead, the democratic predictor achieves 98.8% of the optimal mix of both combiners (it gets 141,731 commands right out of a possible 143,523 obtained by selecting the best combiner on a user-by-user basis). Although this seems somewhat surprising, the democratic combiner's strength lies in its simplicity. By observing the performance of the correlation combiner on larger command sequences (our own data collection), it appears that this combiner is likely to perform better in the long term, but we only have 303,628 commands in the Greenberg dataset. The democratic combiner, however, can simply ignore predictors that are wrong and listen to predictors that are right, improving the signal-to-noise ratio.

If the first 100 commands are ignored as training data, similar results occur, although the correlation combiner becomes the best combiner for a few more novice users. Presumably this is simply due to the correlation combiner "warming up," something the democratic combiner does not need as long to do. Since the macroaverage is a reflection of the final results for users, it is not surprising to see that if ranked by macroaverage, the correlation combiner is the better combiner when ignoring the first 100 commands as training data.

The following may be observed from this:

1. The democratic combiner provides the globally best results when the first 100 command predictions are not ignored as training.

2. The correlation combiner performs best for users with minimal `Unix` experience.

Best Combiner	Non-Programmers	Novice Programmers	Experienced Programmers	Computer Scientists
Correlation	5	2	1	1
Democratic	20	53	35	51

Table 5.8: Breakdown of best combiner to user type

Best Combiner	Accuracy		Number of Users	Commands Right	Total Commands
	Microaverage	Macroaverage			
Correlation	41.75%	47.17%	14	1998	4786
Democratic	47.56%	46.03%	154	134153	282042

Table 5.9: Average accuracy where the given combiner is the best combiner, ignoring training data

5.3 Using All Predictors

To this point, we have determined that we can pick a single best predictor and obtain over 45% macroaverage accuracy. We have also determined that for some users, the best combiner is the correlation combiner, but for most, the democratic combiner is better. However, that is when the optimal combination of predictors is different for each user. For optimal results, we want to have a pool of experts and use an algorithm to determine what the best combination is.

First, we examine what happens when the correlation combiner is used with all the experts. This is our original mixture of experts model put to the test. This model obtains almost 42% macroaverage accuracy (42.5% if the answer to the first 100 commands per user are ignored, i.e., training data) and almost 40% microaverage accuracy (just over 40% if the first 100 commands per user are ignored for training). This is actually worse than simply forcing all users to use the AUR string predictor. As mentioned above, while this is likely the best combiner *in the limit*, it is not with our limited source of data.

Next, the democratic combiner is evaluated. We immediately see that it achieves over 46.5% microaverage accuracy (46.7% without training data) and 48.3% macroaverage accuracy (49% without training data). Clearly, this combination of experts provides better performance than any single expert. As mentioned in Section 5.2, the simplicity of the democratic combiner allows it to easily surpass the correlation combiner.

Finally, we once again explore how results change when the best is selected, in this case, the best combiner for each user. The results here are identical to the democratic combiner when no training

Best Combiner	Non-Programmers	Novice Programmers	Experienced Programmers	Computer Scientists
Correlation	7	2	2	3
Democratic	18	53	34	49

Table 5.10: Breakdown of best combiner to user type, ignoring training data

	Non-Programmers	Novice Programmers	Experienced Programmers	Computer Scientists	Overall
Microaverage	37.26%	57.24%	36.53%	31.69%	39.87%
Macroaverage	38.06%	57.93%	35.15%	30.90%	41.73%
Commands Right	9541	44320	27366	39831	121058

Table 5.11: Accuracy when the correlation combiner and all predictors are used

	Non-Programmers	Novice Programmers	Experienced Programmers	Computer Scientists	Overall
Microaverage	37.81%	57.60%	36.89%	31.90%	40.06%
Macroaverage	41.05%	58.52%	35.65%	31.00%	42.50%
Commands Right	8737	41428	26308	38432	114905

Table 5.12: Accuracy when the correlation and all predictors are used, ignoring training data

	Non-Programmers	Novice Programmers	Experienced Programmers	Computer Scientists	Overall
Microaverage	44.19%	60.76%	44.23%	39.59%	46.52%
Macroaverage	44.68%	61.95%	43.37%	39.07%	48.32%
Commands Right	11316	47039	33132	49763	141250

Table 5.13: Accuracy when the the democratic combiner and all predictors are used

	Non-Programmers	Novice Programmers	Experienced Programmers	Computer Scientists	Overall
Microaverage	44.68%	60.92%	44.58%	39.84%	46.69%
Macroaverage	47.24%	62.29%	43.91%	39.34%	49.01%
Commands Right	10325	43814	31789	48004	133932

Table 5.14: Accuracy when the the democratic combiner and all predictors are used, ignoring training data

	Non-Programmers	Novice Programmers	Experienced Programmers	Computer Scientists	Overall
Microaverage	44.19%	60.76%	44.23%	39.59%	46.52%
Macroaverage	44.68%	61.95%	43.37%	39.07%	48.32%
Commands Right	11316	47039	33132	49763	141250

Table 5.15: Accuracy when the best combiner and all predictors are used

	Non-Programmers	Novice Programmers	Experienced Programmers	Computer Scientists	Overall
Microaverage	44.70%	60.92%	44.58%	39.84%	46.70%
Macroaverage	47.48%	62.29%	43.91%	39.34%	49.04%
Commands Right	10330	43814	31789	48004	133937

Table 5.16: Accuracy then the best combiner and all predictors are used, ignoring training data

data is ignored, and marginally better when training data is ignored (5 more commands out of over 300,000 are correct). This is because two users (both non-programmers with very few commands, one of which is the user with the least total number of commands, 132 , the other has 244 commands) switch to the correlation combiner as the best combiner. This further reinforces the strength of the democratic combiner.

5.4 Omniscience at Work: Gods in Training

Given those results, we can then speculate about some sense of omniscience: what if we knew the best predictor for each user *in advance*? If the best predictor is selected on a user by user basis, we obtain a macroaverage accuracy of 45.76% and microaverage accuracy of 43.26%. A breakdown of this can be seen in Table 5.3. Furthermore, if the first 100 commands are ignored as training data, the macroaverage is 46.55% and the microaverage is 43.39% (the breakdown can be seen in Table 5.5). Although the distribution of best predictor moves around slightly in this case, the best predictor remains the AUR string predictor. Finally, if all combinations of predictors and combiners are considered, the microaverage for the Greenberg data is 47.27% and the macroaverage is 49.37% (see Table 5.17). Again, ignoring the first 100 commands as training gives 47.47% as the microaverage and 50.29% as the macroaverage (see Table 5.18). Given our limited omniscience, we can still benefit from our mixture of experts approach!

	Non-Programmers	Novice Programmers	Experienced Programmers	Computer Scientists	Overall
Microaverage	45.33%	61.50%	45.01%	40.24%	47.27%
Macroaverage	46.71%	62.83%	44.30%	39.94%	49.37%
Commands Right	11609	47616	33717	50581	143523

Table 5.17: Accuracy when the best combiner and best predictors are used

	Non-Programmers	Novice Programmers	Experienced Programmers	Computer Scientists	Overall
Microaverage	45.87%	61.72%	45.39%	40.50%	47.47%
Macroaverage	50.13%	63.30%	44.95%	30.29%	50.29%
Commands Right	10600	44391	32365	48795	136151

Table 5.18: Accuracy when the best combiner and best predictors are used, ignoring training data.

Chapter 6

Future work

"Failure is the opportunity to begin again more intelligently." —Henry Ford

There are multiple paths that future research could follow. First, although we appear to have gleaned all the information from the Greenberg dataset, the predictors look at a very small subset of the attributes available in the data we collected. Although this data has now been collected, its full potential of it has not been exploited. For instance, the current directory the user is in, the phase of the moon[1], how long a command runs, and the time between commands are all available to us, but not currently used. However, adding more attributes for the AUR predictors to use is not enough. Rather, intelligent use of this information is crucial to avoid the curse of dimensionality.

Davison and Hirsh do put forward one interesting idea for future work: automatic execution of non-destructive commands (or at least ones that could be undone)[6]. While this may not be possible, the ability to classify commands as "auto-execute" could be useful for some users. For instance, many people run "`ls`" (list files) after changing directories, as a way of orienting themselves. This is a non-harmful command requiring no "undo."

Although we produced a command shell capable of predicting command lines, we have also considered the possibility of extracting useful command macros. Informal polling of users has shown that automatic alias creation would be a valuable asset for most users. The parsing mechanism in Section 4.3 can be extended to build up macros. A brief attempt to extract macros with those rules resulted in simple commands strung together with all arguments intact (i.e., unparsed). The creation of generic, useful commands requires advances on two fronts. First, the parsing component (Section 4.3) must be improved. If parsing patterns could be learned from observation, this could greatly improve both predictors and the macro generation. Second, an actual macro extraction algorithm should be implemented, as the current method simply reports frequent chains (length three or more) of parsed commands.

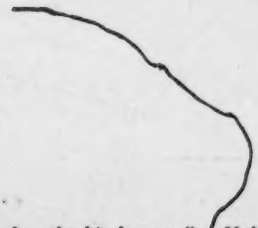
[1] While the phase of the moon may not be relevant to some users, anecdotal experience in the real world would indicate that some events are correlated with the phase of the moon.

On a different note, perhaps this thesis work could be integrated with other projects. UC differs from our work in that it parses user questions and tries to determine the user goal, and provides English descriptions of how to achieve this goal[2]. A merging of that work with our research could see the parsing of commands into representations appropriate for goal recognition. Although much more advanced than our current work, it could provide valuable insight into user modes and long-term goals.

Finally, and most importantly, this work can be applied to *handheld computers*, such as the PalmOS devices. Some have wireless networking capabilities, and `Unix` command lines are difficult to enter without a keyboard. If command lines can be accurately predicted and presented in a list, the time taken to enter a command could be a few seconds rather than a minute. Even more useful would be a program that watches user activity and learns what words come after other words, both application-specific and globally. There are currently products that provide a pop-up list of the most likely words, but they require the user to add words manually and only update themselves when the user selects words from the list (not when the user enters the words manually). An application that learns quickly and predicts efficiently has great potential.

Chapter 7

Conclusion



"Good judgment comes from experience.

Unfortunately, the experience usually comes from bad judgment." —*Unknown*

We present an improvement to the `Unix` command line interface: predicting the next command. We do so by silently observing command lines typed by the user and making a fresh set of predictions after each one of these command lines. The manner in which we observe the user and present predictions is unobtrusive: the user may take advantage of the predictions or she may choose to ignore them with no ill-effect.

Building upon the work by Davison and Hirsh, who predicted only the command stubs [11, 7, 6, 8], we predict the entire command line. To improve prediction accuracy, we use a "mixture of experts" model. This model allows us to have multiple predictors simultaneously and automatically how strongly to weight each one. We have a variety of techniques employed in our predictors: the AUR algorithm allows us to estimate probabilities and track a changing distribution; parsing identifies latent patterns in command lines and provides the ability to synthesize new commands; the command line history can be weighted to favor recent commands; and the last five commands simply predicts those commands.

This combination of complexity and simplicity produces better results than any one predictor. The best single predictor obtains 46% macroaverage accuracy (that is, the average of averages), while the "mixture of experts" model obtains 49%. It is important that when predicting command lines that we be as accurate as possible: the more frequently predictions are correct, the more users will trust our system. After two years and many approaches, we seem to have extracted all useful information from the Greenberg dataset.

Although continued improvements with the Greenberg dataset are unlikely, given the limited attributes and limited sample size (some users have less than 200 command lines), we believe that the additional attributes collected in our dataset can improve prediction accuracy. Thus, while no

significant progress is expected on the Greenberg dataset, it is still possible to improve algorithm performance on alternative datasets. Furthermore, with our dataset it is possible to identify interactions between overlapping command shell sessions. This can be exploited for long-term planning.

These predictions are superior to deployed alternatives: we do not have to educate users beyond telling them "press F1 to run the prediction assigned to F1." The deployed solutions such as `makefiles`, shell scripts and aliases all require users to be trained, or at least have a reference manual. Thus, it is easier to use our system. More importantly, we can inter-operate with the deployed solutions, allowing the best of both worlds.

Other people have tried to predict `Unix` commands lines before. Some have limited their task to simply predicting the command, neglecting the arguments. Some of these "solutions" have often been bulky, requiring a custom operating system, or requiring the "glue" to be Emacs (a bulky program in itself). Yet other solutions do not perform adequately in real-time. Our solution is light-weight in memory usage, runs quickly, and works in any standard `Unix` environment. Our predictions are able to predict not only commands typed before, but able to synthesize new commands from observed patterns (other work was capable of synthesizing new command, but suffered from the need for a custom operating system).

Finally, the application of this thesis work need not be limited to `Unix` command lines. While the hand-coded parsing routines have limited application, the AUR prediction algorithm and the mixture of experts approach can be applied to other areas, such as the rapidly growing field of mobile computing. Our efficient use of resources, through our requirements of realtime performance, makes this an easy step. This is an application of making computers easier to use without the need for faster, bigger, stronger hardware.

Bibliography

[1] Sanjay Bhansali and Mehdi T. Harandi. Synthesis of unix programs using derivational analogy. Technical Report KSL-92-02, Knowledge System Laboratory, Stanford University, Palo Alto, CA, 1992.

[2] David Ngi Chin. *Intelligent Agents as a Basis for Natural Language Interfaces*. PhD thesis, Computer Science Division, Univesity of California, Berkeley, 1987. Also available as Technical Report UCB/CSD 88/396.

[3] Allen Cypher. Eager: Programming repetitive tasks by example. In *Proceedings of CHI'91 Conference on Human Factors in Computing Systems*, pages 33–39. Association for Computing Machinery, April 1991.

[4] Allen Cypher. Eager: Programming repetitive tasks by demonstration. In *Watch What I Do: Programming by Demonstration*, chapter 9. The MIT Press, Cambridge, Massachusetts, 1993.

[5] John J. Darragh and Ian H. Witten. *The Reactive Keyboard*. Cambridge Series on Human-Computer Interaction. Cambridge University Press, New York, New York, 1992.

[6] Brian D. Davison and Haym Hirsh. Experiments in unix command prediction. Technical Report ML-TR-41, Department of Computer Science, Rutgers University, August 1997.

[7] Brian D. Davison and Haym Hirsh. Toward an adaptive command line interface. In *Advances in Human Factors/Ergonomics: Design of Computing Sytems: Social and Ergonomic Considarations*, pages 505–508. Elsevier, 1997.

[8] Brian D. Davison and Haym Hirsh. Predicting sequences of user actions. In *Predicting the Future: AI Approaches to Time-Series Problems*, number WS-98-07, pages 5–12. American Association for Artificial Intelligence, AAAI Press, July 1998.

[9] Saul Greenberg. Using unix: Collected traces of 168 users. Research Report 88/333/45, Department of Computer Science, University of Calgary, Calgary, Alberta, 1988.

[10] Saul Greenberg and Ian H. Witten. How users repeat their actions on computers: principles for design of history mechanism. In *Conference Proceedinds on Human Factors in Computing Systems*, pages 171–178. ACM Press, May 1998.

[11] Haym Hirsh and Brian D. Davison. An adaptive unix command-line assistant. In *Proceedings of the First International Conference on Autonomous Agents*, pages 542–543. Association for Computing Machinery, ACM Press, February 1997.

[12] Thomas Jacob and Benjamin Korvemaker. Adaptive user interfaces: CMPUT 651 project write-up. Class project, December 1998.

[13] Pat Langley. User modeling in adaptive interfaces. In *Proceedings of the Seventh International Conference (UM'99)*, pages 357–370. User Modeling, Springer Wien New York, June 1999. Invited Speaker.

[14] Yezdi Lashkari, Max Metral, and Pattie Maes. Collaborative interface agents. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, pages 444–450. American Association for Artificial Intelligence, AAAI Press, August 1994.

[15] Toshiuki Masui and Ken Nakayama. Repeat and predict — two keys to efficient text editting. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'94)*, pages 118–123. Association for Computing Machinery, ACM Press, April 1994.

[16] Gordon Moore, 1964. http://www.tuxedo.org/ esr/jargon/html/entry/Moore's-Law.html.

[17] Hiroshi Motoda, Takashi Washio, Toshihiro Kayama, and Kenichi Yoshida. Extracting behavioral patterns from relational history data, 1997. Machine Learning for User Modelling Workshop at the Sixth International Conference on User Modeling (UM'97).

[18] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[19] J. C. Schlimmer and L. A. Hermens. Software agents: Completing patterns and constructing user interfaces. *Journal of Artificial Intelligence Research*, 1:61–89, Nov 1993.

[20] Robert Wilensky, David Ngi Chin, Marc Luria, James H. Martin, James Mayfield, and Dekai Wu. The berkeley unix consultant project. Technical Report UCB/CSD 89/520, Computer Science Division, Univesity of California, Berkeley, 1989.

[21] Kenichi Yoshida and Hiroshi Motoda. Automated user modeling for intelligent interface. *International Journal of Human-Computer Interaction*, 8(3):237–258, 1996.

Index

Adaptive User Interface, 10
AddChild
    AUR-Predictor, 70
alias, 8, 9
Alpha Update Rule, 3, 17, 21, 25, 32, 33, 35–40, 42, 44, 46, 47, 54, 66, 69
AlphaUpdate
    AUR-Predictor, 70
APU, 18, *see* [1]
AUI, *see* Adaptive User Interface
AUR, *see* Alpha Update Rule
AUR-Predictor
    AddChild, 70
    AlphaUpdate, 70
    CreateChildren, 71
    Predict, 72
auto-completion, 12
auto-execute, 44

Backwards Predictor, 65

C4.5, 13, *see* [18], 14, 17
ClipBoard, *see* [17, 21], 13
collaboration, 17
command, 52
    argument, 52
    option, 52
conditional probability table, 14
CPT, *see* Conditional Probabilty Table
CreateChildren
    AUR-Predictor, 71
Cycle Predictor, 67, 68

default distribution, 14

Eager, 11, *see* [3, 4]
Emacs, 12, 47

fail-soft, 17, 18
Feed
    Tuples, 69
file I/O, 13
frobnicate, *see* xfrobnicate, 53

GBI, *see* Graph Based Induction
Graph Based Induction, 13
Graphical User Interface, 7, 11, 13, 52
GUI, *see* Graphical User Interface

handheld computers, 45, *see* mobile computing
HCI, *see* Human-Computer Interaction
History Predictor, 64
History Weighted Predictor, 64, 65
Human-Computer Interaction, 57
Hypercard, 11

Ideal Online Learning Algorithm, 14
ilash, 14
Incremental Probablistic Action Modeling, 14, 17, 23, 25, 27, 30, 37
Inductive Learning Apprentice SHell, *see* ilash

KNOME, 17, *see* [2, 20]

Last-N Predictor, 65
learning
    offline, *see* offline learning
    online, *see* online learning
Linux, 12
Lisp, 11, 18
Longest Matching Prefix, 13

Machine Learning, 14, 18, 19, 30
Macintosh, 11
macro, 8, 44
macroaverage, 13, 14, 17, *see* microaverage
make, 8
makefile, 8
meta-knowledge, 21
microaverage, 13, 14, 17, *see* macroaverage
mobile computing, 47
Mode Predictor, 66, 67
Moore's Law, 10
Most Frequent Command, 13
Most Recent Command, 13

Natural Language Processing, 18, 19

offline learning, 13, 14, *see* online learning
Omniscient, 13
online learning, 10, 13, *see* offline learning

PalmOS, 45
parse, 45
parsed command line, 33
parser, 32
Parsing, 31, 34
parsing, 31–33, 44, 45
pen-based computing, 11
Predict
    AUR-Predictor, 72
programming by demonstration, *see* programming by example
programming by example, 11, 14

Reactive Keyboard, 12, *see* [5]
real-time, 20

repeat and predict, 12
RK, *see* Reactive Keyboard

script, 9
scripts, 9
shell, 8, 9, 52, 61
Solaris, 12
Starve
    Tuples, 69
Stub Predictor, 65, 66

tcsh, 13
Tuples
    Feed, 69
    Starve, 69

UC, *see* Unix Consultant
Unix Consultant, 17, *see* [2, 20], 18, 45

Windows, 12

xfrobnicate, 7, *see* frobnicate, 53

zsh, 2, 61

Appendix A

Unix Command Line Structure

In the `Unix` command line environment, a program called a *shell* is used to enter commands. Although a GUI can be used to perform some tasks, unless a command (or action) is incredibly simple, parameters must be specified. Specifying parameters with a GUI usually involves highlighting something (text, pictures, etc.) with the mouse, checking boxes and selecting items from pull-down lists. In contrast, multiple parameters are specified easily with a `Unix` command line, but doing so frequently involve complex combinations of dashes and alphanumeric sequences (e.g., "`ls -latr`" lists files with details, increasingly sorted by time). Consequently, novice users are frequently traumatized when they encounter a `Unix` command line.

A simple command line can be typically broken into three parts: the *command*, the *command options*, and the *command arguments*. For example, a user might type the command line

```
mail -s 'have you seen my printer?' fred
```

This will run the command "mail" with the subject "have you seen my printer?" and the recipient "fred." The command option "`-s`" indicates to "`mail`" that the next argument (have you seen my printer?) should be used as the default subject. The subject must be enclosed in quotes because it has spaces in it, and without spaces, "`mail`" would not know where the subject ended and the recipient list started (in fact, it assumes that unless told otherwise, arguments on the command line are recipients). In recap:

1. `mail` is a command
2. `-s` is a command option
3. `'have you seen my printer?'` is a command argument, given special meaning by the preceding -s
4. `fred` is a command argument

While knowledge of `Unix` commands is not crucial for understanding this thesis, it is important to understand that most commands are constructed in this manner. For a further example, in Chapter 2, two command sequences are given: Fred writing a thesis and Wilma writing a program. Table A.1 and Table A.2 provide breakdowns of those command sequences.

command	parameter	meaning
`vi`	`thesis.tex`	the file Fred is editing
`latex`	`thesis.tex`	the LaTeX file to be processed into a device-independent file
`dvips`	`-q`	tells `dvips` to only report errors
	`thesis.dvi`	the file to be processed into a PostScript file
	`-t letter`	tells dvips to use letter-size paper
	`-o thesis.ps`	is the destination PostScript file
`ghostview`	`thesis.ps`	is the PostScript file to be viewed
`mail`	`-s 'I need latex help'`	specifies the subject will be "I need latex help"
	`wilma`	is the intended recipient of the email message

Table A.1: Breakdown of the commands Fred types in Section 2.1

command	parameter	meaning
`vi`	`frobnicate.c`	file Wilma is editing
	`xfrobnicate.c`	file Wilma is editing
`cc`	`-c`	tells `cc` to generate object code
	`frobnicate.c`	file to generate object code from
	`xfrobnicate.c`	file to generate object code from
	`-I/usr/X11R6/include`	tells `cc` where to look for additional files
`ld`	`-o xfrobnicate`	tells `ld` where to put the destination file
	`frobnicate.o`	object code file
	`xfrobnicate.o`	object code file
	`-L/usr/X11R6/lib`	tells `ld` where to look for additional files
	`-lXt`	tells `ld` to use the Xt library when linking
	`-lX11`	tells `ld` to use the X11 library when linking

Table A.2: Breakdown of the commands Wilma types in Section 2.2

Appendix B

AUR Predictor Example

This is a simple example of how the AUR predictor in Section 4.2 updates its prediction tables. The command sequence can be seen in Figure B.1. The prediction table evolves over time in Tables B.1 to B.6.

time	
0	`vi thesis.tex`
1	`latex thesis.tex`
2	`dvips thesis.dvi -o thesis.ps`
3	`gv thesis.ps`
4	`vi thesis.tex`
5	`lpr thesis.ps`
6	`vi thesis.tex`
7	`latex thesis.tex`
8	`dvips thesis.dvi -o thesis.ps`
9	`gv thesis.ps`
10	`lpr thesis.ps`
11	`rm thesis.ps`
12	`cd src`
13	`vi thesis.c`
14	`cc thesis.c -o thesis`
15	`./thesis`
16	`vi thesis.c`
17	`cc thesis.c -o thesis`
18	`./thesis`
19	`vi thesis.c`
20	`cc thesis.c -o thesis`
21	`./thesis`
22	`rm thesis`
23	`cd ..`

Figure B.1: Sample command sequence

$command_t$	Probability
`dvips thesis.dvi -o thesis.ps`	4.75%
`gv thesis.ps`	5.00%
`latex thesis.tex`	4.51%
`vi thesis.tex`	85.74%

Table B.1: Probability table at time 3

$command_t$	Probability
`dvips thesis.dvi -o thesis.ps`	4.51%
`gv thesis.ps`	4.75%
`latex thesis.tex`	4.29%
`vi thesis.tex.`	86.45%

Table B.2: Probability table at time 4

$command_t$	Probability
`dvips thesis.dvi -o thesis.ps`	4.29%
`gv thesis.ps`	4.51%
`latex thesis.tex`	4.07%
`lpr thesis.ps`	5.00%
`vi thesis.tex`	82.13%

Table B.3: Probability table at time 5

$command_t$	Probability
`dvips thesis.dvi -o thesis.ps`	4.07%
`gv thesis.ps`	4.29%
`latex thesis.tex`	3.87%
`lpr thesis.ps`	4.75%
`vi thesis.tex`	83.02%

Table B.4: Probability table at time 6

$command_t$	Probability
`dvips thesis.dvi -o thesis.ps`	7.44%
`gv thesis.ps`	7.83%
`latex thesis.tex`	7.07%
`lpr thesis.ps`	8.43%
`rm thesis.ps`	5.00%
`vi thesis.tex`	64.24%

Table B.5: Probability table at time 11

$command_t$	Probability
`./thesis`	11.70%
`cc thesis -o thesis`	11.11%
`cd ..`	5.00%
`cd src`	2.84%
`dvips thesis.dvi -o thesis.ps`	4.02%
`gv thesis.ps`	4.23%
`latex thesis.tex`	3.82%
`lpr thesis.ps`	4.55%
`rm thesis`	4.75%
`rm thesis.ps`	2.70%
`vi thesis.c`	10.56%
`vi thesis.tex`	34.71%

Table B.6: Probability table at time 23

Appendix C

Evaluation of Command List Size

The reasons for selecting $n = 5$ are fourfold: Davison and Hirsh experimentally determined five was optimal (less than five provided poorer results, more than five provided essentially the same results); our own initial experiments supported Davison and Hirsh's results; Human-Computer Interaction (HCI) literature states the human cognitive limit is 7 ± 2; and our own brief testing showed signs of delay with more than five command lines.

We explored the possibility of predicting more than five command lines at a time. We also explored what the effects of inverting the list of predictions (displaying the least likely command first, and the most likely command last). After a brief study, there appears to be a problem with presenting *users* with more than five commands, even if improved accuracy could be obtained by doing so. With this evidence, our early experience that predicting more than five command provides no significant advantage, and Davison and Hirsh's similar finding, we believe that five commands is the optimal number. A summary of our results can be seen in Tables C.1–C.4.

	Number of Commands Presented									
User	1	2	3	4	5	6	7	8	9	10
Ben	1.24	1.31	1.56	1.53	1.83	1.77	2.08	2.03	2.09	2.38
Chris	1.62	1.45	1.41	1.45	1.67	2.02	2.15	2.59	2.54	3.17
Russ	2.44	2.99	2.21	2.49	3.28	2.59	3.77	378	3.76	4.09
Stef	1.91	2.27	2.05	4.58	2.64	2.71	2.79	3.21	3.57	3.84

Table C.1: Average time in seconds taken to select a command for n commands presented.
See Figure C.1.

	Number of Commands Presented									
User	1	2	3	4	5	6	7	8	9	10
Ben	1.34	1.27	1.37	1.48	1.90	1.77	2.72	2.12	2.03	1.97
Chris	1.03	1.10	1.24	1.40	1.51	1.66	2.13	1.85	1.86	2.11

Table C.2: Average time in seconds taken to select a command for n commands presented when command list is inverted.

See Figure C.2.

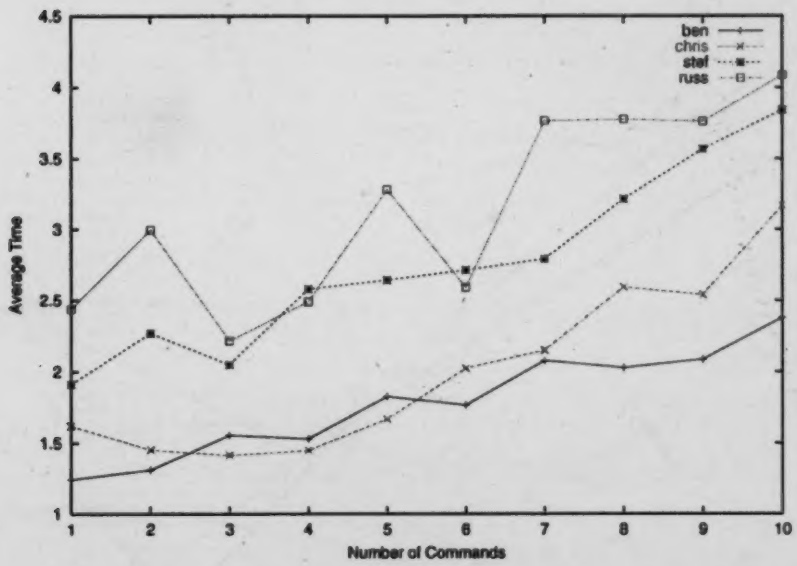


Figure C.1: Average time in seconds taken to select a command for n commands presented. See Table C.1.

	Number of Commands Presented									
User	1	2	3	4	5	6	7	8	9	10
Ben	1.37	1.44	1.65	1.79	1.86	1.84	2.19	2.27	2.29	2.33
Chris	1.10	1.45	1.54	1.77	1.66	1.83	2.00	2.13	2.23	2.35

Table C.3: Average time in seconds taken to select a command for n commands presented. See Figure C.3.

	Number of Commands Presented									
User	1	2	3	4	5	6	7	8	9	10
Chris	0.89	1.17	1.32	1.47	1.48	1.67	1.74	1.93	1.99	2.08

Table C.4: Average time in seconds taken to select a command for n commands presented when command list is inverted.

See Figure C.4.

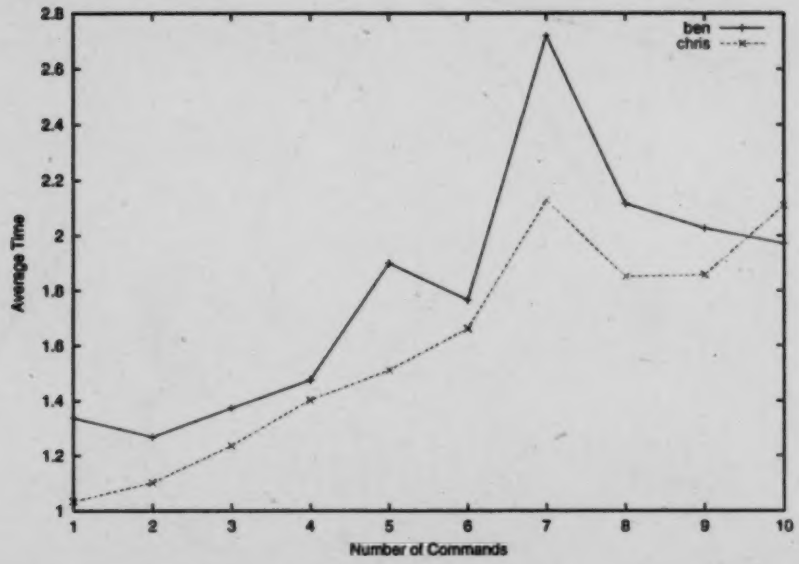


Figure C.2: Average time in seconds taken to select a command for n commands presented when command list is inverted.

See Table C.2.

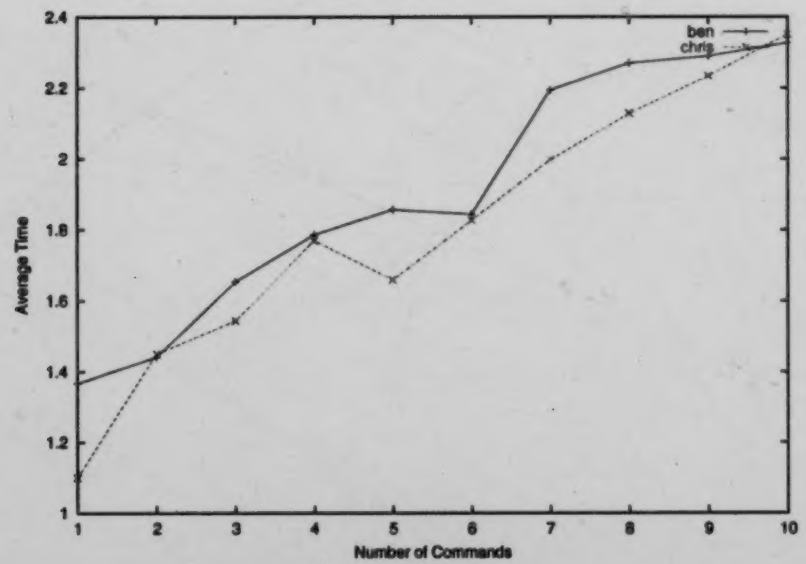


Figure C.3: Average time in seconds taken to select a command for n commands presented.
See Table C.3.

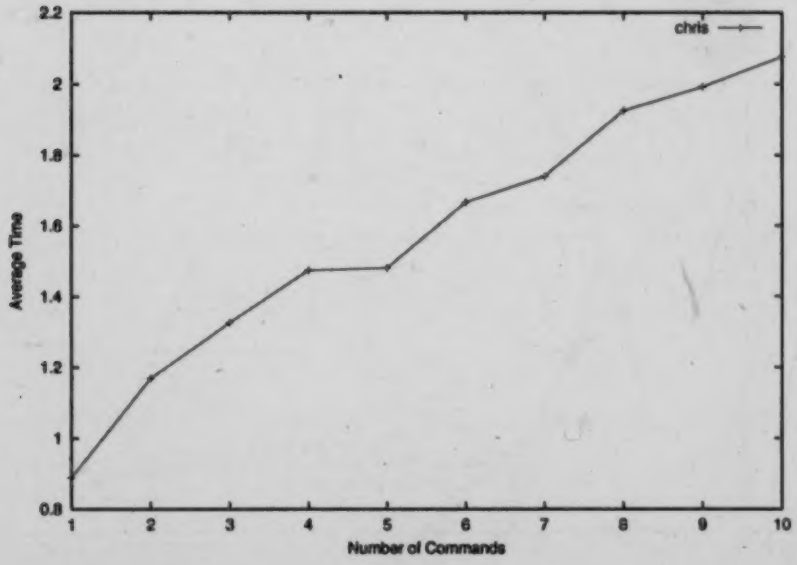


Figure C.4: Average time in seconds taken to select a command for n commands presented when command list is inverted.

See Table C.4.

Appendix D

Pictures of Our Implementation

Throughout the course of this thesis work, our goal was to have a predictive shell that could be used with minimal effort on the part of the user. During the algorithm development, we produced two different visual interfaces, both of which were used the same way: press an F-key to use a predicted command. Additionally, both use the ZSH command shell, selected initially for its clean source code for easy modification. As the prediction methods evolved, we realized that source code modification was unnecessary, as ZSH's scripting language is extremely powerful.

D.1 Using the Title Bar

Our first method of presenting predictions to users was to modify the title of the `xterm`, a box that runs the shell. The title can be easily changed after each command, so it is easy for the user to know what the current mappings of the F-keys are. An enhanced screenshot of such an interface can be seen in Figure D.1, where th e commands are highlighted with a green background for visibility, and the predictions can be seen red in the title bar at the top. As can be seen, there is little room in the title bar for long predictions. Since this situation is made worse when larger fonts are used, we considered alternate methods of presenting the predictions.

D.2 Using a Separate GUI

We briefly considered displaying the predictions inside the xterm, before the prompt was displayed. However, doing so increases the intrusiveness of the predictions, something we want to minimize. Alternatively, we chose to create a second window that could be displayed above or below the xterm, increasing the display flexibility for the user, while providing use with additional display space. Rather than displaying all predictions on one line, we display each prediction on a separate line. A screenshot of this can be seen in Figure D.2. Although additional screen real-estate is lost to this display, we believe it to be a superior method.

F1 [dvips peqnp] F2 [bibtex peqnp] F3 [vi peqnp.tex] F4 [latex peqnp] F5 [ls]
(/usr/lib/texmf/texmf/tex/latex/base/article.cls
Document Class: article 1996/10/31 v1.3u Standard LaTeX document class
(/usr/lib/texmf/texmf/tex/latex/base/size10.clo)) (peqnp.aux)
(/usr/lib/texmf/texmf/tex/latex/base/omscmr.fd) [1] (peqnp.aux))
Output written on peqnp.dvi (1 page, 804 bytes).
Transcript written on peqnp.log.
~/tmp/z
This is dvipsk 5.58f Copyright 1986, 1994 Radical Eye Software
' TeX output 1999.01.18:2315' -> peqnp.ps
<tex.pro>. [1]
~/tmp/z
~/tmp/z
~/tmp/z
This is TeX, Version 3.14159 (C version 6.1)
(peqnp.tex
LaTeX2e <1996/12/01> patch level 1
Babel <v3.6h> and hyphenation patterns for american, german, loaded.
(/usr/lib/texmf/texmf/tex/latex/base/article.cls
Document Class: article 1996/10/31 v1.3u Standard LaTeX document class
(/usr/lib/texmf/texmf/tex/latex/base/size10.clo)) (peqnp.aux)
(/usr/lib/texmf/texmf/tex/latex/base/omscmr.fd) [1] (peqnp.aux))
Output written on peqnp.dvi (1 page, 804 bytes).
Transcript written on peqnp.log.
~/tmp/z

Figure D.1: Title bar interface screenshot

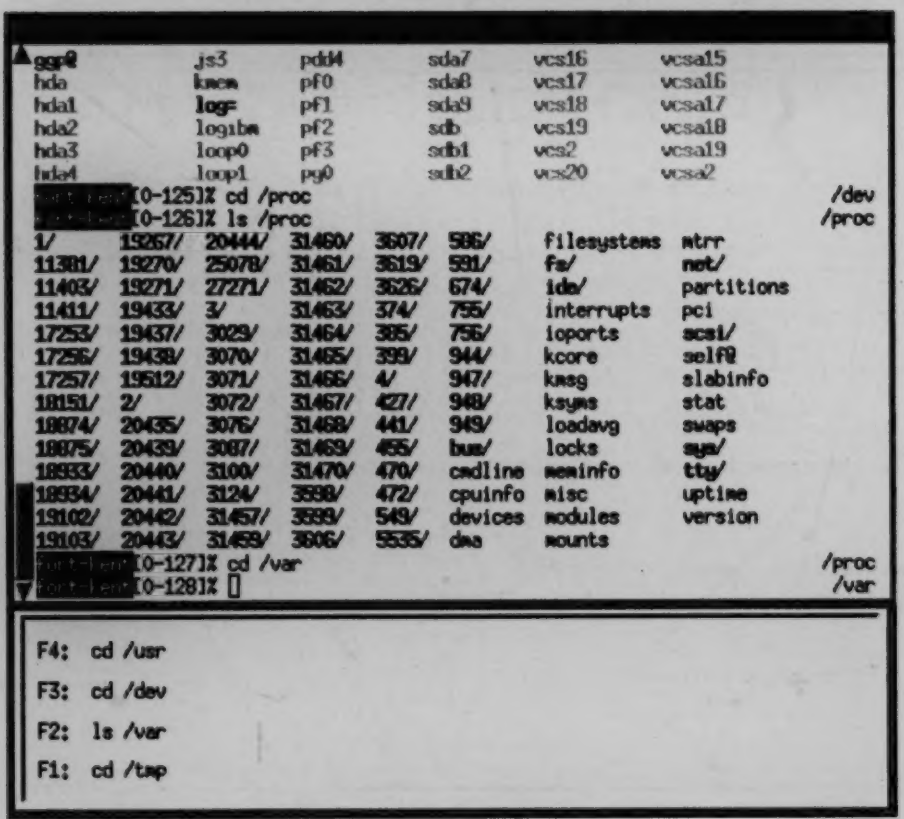


Figure D.2: Separate display screenshot

Appendix E

Additional Predictors

These predictors were constructed by Christopher Thompson as part of a related research project.

E.1 History Predictor

See Algorithm E.1. This predictor looks back at the last $n = 100$ commands and predicts based on frequency. The rationale behind such a predictor is that a user spends a lot of time doing the same thing over and over. It is much more likely that the next command will be a command that was executed recently rather than a command from long in the past. This combiner does not weight more recent commands higher, unlike the predictor in Section E.2. Each command line "votes" on what the next one will be, with each vote worth the same amount (i.e., $P(\chi) = \frac{\#\ of\ commands = \chi}{\#\ of\ commands}$ for a window no greater than 100 commands).

Algorithm E.1 History Predictor

1: $LOOKBACK \leftarrow 100$
2: $length \leftarrow |history|$
3: $start \leftarrow \max(length - LOOKBACK, 0)$
4: **for** $i = start \ldots length - 1$ **do**
5: $\quad score_{history_i} + +$

where $history$ is the command line history, and $history_i$ is an individual command line.

E.2 History Weighted Predictor

See Algorithm E.2. This predictor looks back at the last N commands and predicts based on frequency, in a manner similar to the history predictor in Section E.1. The rationale behind this predictor is the same: the recent past is a better indicator than the long-term past. However, this combiner weights more recent commands higher. Each command line "votes" on what the next one will be, with each vote worth as much as the weighting. The specific weighting function is as follows:

1. For histories longer than $n = 100$, look at only the last n entries, otherwise, look at all entries.
2. Each line contributes $\frac{position \times 2}{length \times (length+1)}$ weighting.

Algorithm E.2 History Weighted Predictor

```
1: LOOKBACK ← 100
2: length ← |history|
3: start ← max(length − LOOKBACK, 0)
4: divisor ← (length − start) × (length−start+1)/2
5: for i = start ... length − 1 do
6:     score_{history_i} += (i−start+1)/divisor
```

E.3 Last-N Predictor

This predictor simply predicts the last $n = 5$ unique command lines, looking back as far as needed. The rationale behind this predictor is even simpler: a user will often use the up-arrow to re-enter a command. While there is minimal intelligence behind such a predictor, its performance provides a reasonable baseline for other predictors to beat.

E.4 Backwards Predictor

See Algorithm E.3. This predictor makes a prediction based on context. The rationale behind this predictor is that although certain groups of commands naturally fall together, there is also a lot of noise. For example, perhaps a user normally edits a source file, then compiles, and repeats. However, one time, the user takes a look at how many lines of code is in the source file. Still, the computer should be able to guess that this was just noise and make the same prediction again. The backwards predictor keeps a running total of what preceded a given command, weighting the command at $time_{t-1}$ higher than at $time_{t-2}$. When it comes time to make a prediction, it looks at the last two commands the user typed in and selects the commands that have the highest count. For additional robustness against noise, it uses command stubs to predict the command lines.

Algorithm E.3 Backwards Predictor

```
1: PREV_2_VALUE ← 3
2: PREV_1_VALUE ← 5
3: length ← |history|
4: if length < 3 then
5:     return ∅
6: for i = 2 ... length − 1 do
7:     prev_2 ← history_{i−2}
8:     prev_1 ← history_{i−1}
9:     curr ← history_i
10:    graph[curr][prev_2] += PREV_2_VALUE
11:    graph[curr][prev_1] += PREV_1_VALUE
12: prev ← history_{length−2}
13: curr ← history_{length−1}
14: for all item ∈ graph do
15:    score[item_command] += item_{score[prev]} + item_{score[curr]}
16: return score
```

E.5 Stub Predictor

See Algorithm E.4 for both predictors. These predictors have been described as a rather stupid prediction method, simply looking back through the history of commands and providing a probability for each of them based on a stub from the command. The behavior of the algorithm is affected through *getStripped*(). The stub first predictor predicts based on the command stub. The stub last predictor uses the last non-option argument.

Algorithm E.4 Stub Predictor

1: $length \leftarrow |history|$
2: **if** $length < 2$ **then**
3: **return** $\emptyset$
4: $current \leftarrow history_{length-1}$
5: $last \leftarrow history_{length-2}$
6: $currentStripped \leftarrow getStripped(current)$
7: $lastStripped \leftarrow getStripped(last)$
8: **if** $length = 2$ **then**
9: $stats_{lastStripped \rightarrow last++}$
10: $stats_{currentStripped \rightarrow current++}$
11: **if** $\exists item \in stats_{lastStripped} | item_{command} = currentStripped$ **then**
12: $item_{count}++$
13: **else**
14: $stats_{lastStripped} \leftarrow stats_{lastStripped} \cup (currentStripped, 1)$
15: **for all** $item \in stats_{currentStripped}$ **do**
16: $score_{item.command} += item.score$
17: **return** $score$

E.5.1 Stub First Predictor

This predictor uses the frequency of first stub in command line for predictions because a user working on a specific file tends to execute the same commands. This algorithm works by looking at each stub and seeing what comes next. It makes its prediction based on this. For example, consider the sequence of commands "`emacs foo.txt`", "`blarf foo.txt`", "`ls`" "`emacs bar.txt`", "`blarf baz`", "`emacs bar.txt`". The user will most likely type "`blarf baz`" next, since every time "`emacs` <something>" is typed, "`blarf` <something>" follows.

E.5.2 Stub Last Predictor

This predictor uses the frequency of last stub in command line for predictions for similar reasons. However, the algorithm looks at the opposite end of the command line. It should be noted that the stub for a given command line will be the last thing on that line that does not immediately appear to be a switch. For example, the stub for "`foo bar baz -l`" is "`baz`", in contrast with our usual definition of a stub. Similarly, "`foo -al bar`" has "`bar`" for a stub. However, "`foo -l -h`" has two options specified, so "`foo`" is the stub and consequently, this predictor degenerates into a Stub First Predictor for many command lines.

E.6 Mode Predictor

See Algorithm E.5. This predictor makes a prediction based on what mode a user is in, but may stay silent. The rationale behind this predictor is that users tend to repeat the same set of commands at very different times. For example, a user may start up an xterm, check email, check the newsgroups, and finally start Netscape. This sort of pattern occurs often but may be interrupted by long delays of semi-random commands. This algorithm works by predicting a command based on the stubs of the previous two commands typed in. If it has seen that pattern before, it predicts. If it has not, it stays silent. The key difference between this and the AUR predictor described in Section 4.2 is that this one may stay silent, while the AUR predictor has a default table used in new situations. A consequence of this silence is that although it has a high accuracy rate when it predicts, it predicts a much lower number of correct commands compared to other algorithms.

Algorithm E.5 Mode Predictor

```
1: length ← |history|
2: if length < 4 then
3:    return ∅
4: for i = 2 length − 2 do
5:    prev2 ← history_{i−2}.stub
6:    prev1 ← history_{i−1}.stub
7:    curr ← history_i.stub
8:    curr ← history_{i+1}.stub
9:    if ∃item ∈ graph|item.prev2 = prev2 and item.prev1 = prev1 and item.curr =
      curr and item.next = next then
10:      item_count + +
11:   else
12:      graph ← graph ∪ [curr, prev2, prev1, next, 1]
13: prev2 ← history_{length−2}.stub
14: prev1 ← history_{length−1}.stub
15: curr ← history_length.stub
16: for all item ∈ graph|item.prev2 = prev2 and item.prev1 = prev1 and item.curr = curr
    do
17:    score_{item.next} + = item.count
18: return score
```

E.7 Cycle Predictor

See Algorithm E.6. This prediction algorithm is significantly different from the rest. It provides no benefit to historical data analysis, but rather provides a helpful hand during interactive use. Consequently, we have no performance data for it. For example, if a user types "`make foo`" followed by "`gdb foo`", the predictor will predict "`make foo; gdb foo`". It exists in two flavors, one that uses the complete command line, and one that uses command stubs.

E.7.1 Entire Cycle Predictor

This is a macro-generating predictor that uses the entire command line. Quite often, a user ends up typing the same set of commands in over and over. While we can predict each one in turn, we may also be able to predict all of them at once. Any reasonable `Unix` shell allows the user to combine multiple commands with either a semicolon or ampersands separating each. The algorithm builds up a graph of which command followed which, and then searches through this graph, for a chain of commands that begin and end with the current command. If one is found, the entire cycle is predicted as a single command.

E.7.2 Stub Cycle Predictor

This is a macro-generating predictor that uses command stubs. Similar to the predictor in Section E.7.1, it searches for cycles. However, it uses command stubs for the graph, continuing to predict a chain of complete command lines. We are unclear whether or not the use of stubs will provide any significant improvement.

Algorithm E.6 Cycle Predictor

```
 1: LOOKCYCLES ← 128
 2: length ← |history|
 3: if length > 1 then
 4:   for i = max(length − LOOKCYCLES + 1, 1) ... length − 1 do
 5:     current ← history_i
 6:     previous ← history_{i−1}
 7:     if ∃item ∈ graph | item_command = previous, item_count = max then {replaceInGraph}
 8:       if ∃neighbour ∈ item_neighbourlist | neighbour_command = current then
 9:         neighbour_count + +
10:       else
11:         item_neighbourlist ← item_neighbourlist ∪ ⟨current, 1⟩
12:       {addToGraph}
13:       graph_previous ← graph_previous ∪ ⟨current, 1⟩
14: return searchCycles (graph, current, current, count = 4)

 1: searchCycles (graph, current, target, count)
 2: CYCLELENGTH ← 4
 3: results ← ∅
 4: if count = 0 then
 5:   return ∅
 6: if item ∈ graph|item_command = current, item_count = max then
 7:   {Take the most frequent neighbor and look at what follows it!}
 8:   for all neighbour ∈ item_neighbourlist do
 9:     if count < CYCLELENGTH and neighbour_command = target then
10:       results ← results ∪ target
11:     else
12:       deeper ← searchCycles (graph, neighbour_command, target, count − 1)
13:       for all entry ∈ deeper do
14:         results ← results ∪ neighbour_command + ';' + entry
15: return results
```

Appendix F

Algorithms for AUR Predictor

This appendix contains a brief step by step description for key components of the AUR predictor.

Algorithm F.1 Tuples::Feed($historyEntry, command$)

1: $key \leftarrow historyEntry$
2: **if** $key_{command} \in RemovedCommands$ **then**
3: $key_{command} \leftarrow REMOVEDCOMMAND$
4: **if** $key_{exit\ code} \in RemovedExitCodes$ **then**
5: $key_{exit\ code} \leftarrow REMOVEDEXITCODE$
6: **if** $key_{time} \in RemovedTimes$ **then**
7: $key_{time} \leftarrow REMOVEDTIME$
8: **if** $key_{day} \in RemovedDays$ **then**
9: $key_{day} \leftarrow REMOVEDDAY$
10: **if** $\exists t \in data \,|\, t_{key} = key$ **then**
11: $t_{TTL} \leftarrow TTLRESET$
12: $t_{count} \leftarrow t_{count} + 1$
13: **else** {Add a new item}
14: $t_{key} = key$
15: $t_{TTL} \leftarrow TTLRESET$
16: $t_{count} \leftarrow 1$
17: $data \leftarrow data \cup t$

Algorithm F.2 Tuples::Starve($void$)

1: **for all** $t \in data$ **do**
2: $t_{TTL} \leftarrow t_{TTL} - 1$
3: **if** $t_{TTL} < 0$ **then**
4: $data \leftarrow data - t$

Algorithm F.3 AUR-Predictor::AlphaUpdate(*command*)

1: $total \leftarrow 0$
2: **for all** $t \in alphaTable$ **do**
3: $\quad total \leftarrow total + t_{probability}$
4: **for all** $t \in alphaTable$ **do**
5: $\quad t_{probability} \leftarrow t_{probability}/total$
6: $\quad t_{probability} \leftarrow t_{probability} \times \alpha$
7: **if** $\exists t \in alphaTable | t_{command} = command$ **then**
8: $\quad t_{probability} \leftarrow t_{probability} + (1 - \alpha)$
9: **else**
10: $\quad t = \langle command, 1 - \alpha \rangle$
11: $\quad alphaTable \leftarrow alphaTable \cup t$

Algorithm F.4 AUR-Predictor::AddChild(*branch*, *sequence*)

1: $S \leftarrow tuples.EachSampleFor\,(branch)$
2: $total \leftarrow 0$
3: **for all** $s \in S$ **do**
4: $\quad total \leftarrow total + s_{count}$
5: $bootstrapTable \leftarrow \emptyset$
6: **for all** $s \in S$ **do**
7: $\quad bootstrapTable \leftarrow bootstrapTable \cup \langle s_{command}, \frac{s_{count}}{total} \rangle$
8: $c \leftarrow Predictor\,(branch, sequence, bootstrapTable)$
9: $children \leftarrow children \cup c$

Where:
S is a set $\langle command, count \rangle$ of all possible commands that *branch* could predict.
branch is a command, exit code, time of day, or day of week.
If a command line occurs at $time_t$, this branch examines the command line at $time_{t-sequence}$ to predict the next command.
$bootstrapTable = \left\{ \langle s_{next}, \frac{s_{count}}{total} \rangle \middle| s \in S, total = \sum_x^S x_{count} \right\}$
bootstrapTable is a calculated probability table to initialize *alphaTable* for the new child node.
children is the set of all child nodes for this node.

Algorithm F.5 AUR-Predictor::CreateChildren(*history*)

Ordering is: Day of Week, Time of Day, Command, Exit Code, Root

1: $S \leftarrow tuples_t.EachSample()$
2: $size_S \leftarrow tuples_t.TotalSamples$
3: $H_S \leftarrow Entropy(S, size_S)$
4: $S\prime \leftarrow tuples_{t-1}.EachSample()$
5: $size_{S\prime} \leftarrow tuples_{t-1}.TotalSamples$
6: $H_{S\prime} \leftarrow Entropy(S\prime, size_{S\prime})$
7: **if** $TYPE <$ Command **then**
8: $\quad B_C \leftarrow tuples_t.Commands()$ {$\langle command, count\rangle$ in $tuples_t$}
9: **if** $TYPE <$ Exit Code **then**
10: $\quad B_E \leftarrow tuples_t.ErrorCodes()$ {$\langle exit\ code, count\rangle$ in $tuples_t$}
11: **if** $TYPE <$ Time of Day **then**
12: $\quad B_T \leftarrow tuples_t.Times()$ {$\langle time, count\rangle$ in $tuples_t$}
13: $B_{C\prime} \leftarrow tuples_{t-1}.Commands()$ {$\langle command, count\rangle$ in $tuples_{t-1}$}
14: $B_{E\prime} \leftarrow tuples_{t-1}.ExitCodes()$ {$\langle exit\ code, count\rangle$ in $tuples_{t-1}$}
15: $B_{T\prime} \leftarrow tuples_{t-1}.Times()$ {$\langle time, count\rangle$ in $tuples_{t-1}$}
16: $B_D \leftarrow tuples_{t-1}.Days()$ {$\langle day, count\rangle$ in $tuples_{t-1}$}
17: $RemoveNonOptions(B)$
18: **if** $B = \emptyset$ **then**
19: $\quad$ **return**
20: $G \leftarrow Gain(B, S, size_S, H_S)$
21: $best \leftarrow \max G$
22: $AddChild(best, sequence + 1)$

Where:
tuples is a set $\langle\langle command, exit\ code, time, day, count\rangle \Rightarrow predicted\ command\rangle$
S is a set $\langle predicted\ command, count\rangle$, all possible commands to predict from $tuples_t$ (line 1).
Similarly for $S\prime$ and $tuples_{t-1}$.
$size_S = \sum_s^S s.count$, $size_{S\prime} = \sum_s^{S\prime} s.count$
$H_S = Entropy(S, |S|)$, $H_{S\prime} Entropy(S\prime, |S\prime|)$
B_C and $B_{C\prime}$ are sets of $\langle command, count\rangle$, $B_C \subseteq tuples_t$,$B_{C\prime} \subseteq tuples_{t-1}$.
B_E and $B_{E\prime}$ are sets of $\langle exit\ code, count\rangle$, $B_E \subseteq tuples_t$,$B_{E\prime} \subseteq tuples_{t-1}$.
B_T and $B_{T\prime}$ are sets of $\langle time, count\rangle$, $B_T \subseteq tuples_t$,$B_{T\prime} \subseteq tuples_{t-1}$.
B_D is a set of $\langle day, time\rangle$, $B_D \subseteq tuples_{t-1}$.
G is a set of $\langle value, gain\rangle$ representing the gain from splitting on value
$best = \max^{gain}(G)$

Algorithm F.6 AUR-Predictor::Predict($history, scale, incr\_scale$)

```
1:  last ← |history|
2:  me ← last − SEQUENCE
3:  ok ← false
4:  if SEQUENCE = 0 then
5:     ok ← true
6:  else if TYPE = Command and COMMAND = history_me.COMMAND then
7:     ok ← true
8:  else if TYPE = Exit Code and ERROR = history_me.ERROR then
9:     ok ← true
10: else if TYPE = Day of Week and DAY = history_me.DAY then
11:    ok ← true
12: else if TYPE = Time of Day and TIME = history_me.TIME then
13:    ok ← true
14: if ok ≠ true then
15:    return ∅
16: for all child ∈ children do
17:    temp = child.Predict (history, scale × incr_scale, incr_scale) {Recursive}
18:    for all p ∈ temp do
19:       if ∃q ∈ predictions|p_COMMAND = q_COMMAND then
20:          q_PROB ← q_PROB + p_PROB
21:       else
22:          predictions ← predictions ∪ p
23: for all p ∈ alphaTable do
24:    if ∃q ∈ predictions|p_COMMAND = q_COMMAND then
25:       q_PROB ← q_PROB + q_PROB × scale
26:    else
27:       predictions ← predictions ∪ (p × scale)
28: return predictions
```

Appendix G

Data Collection

Although the Greenberg dataset is key in our evaluation, we also have our own data, collected for over a year, although on a smaller set of users. To address the limitations of the Greenberg data, we began our own opt-in data collection, collecting:

- command line
- exit code
- time command started and ended (with sub-second accuracy)
- host identification
- **all** environment variables and shell settings (this includes the current working directory because it is contained in the PWD variable)
- all shell aliases

Further, all data was collected into a multiplexed log file so we have exact sequencing of all commands for all users, even if the system clock on the computers in use became out of synchronization.

The collection of this data is accomplished by using two functions provided by ZSH and a small helper program. This allows us to instrument ZSH without modifying or even examining the source. The helper program simply observes the current time and sends a message (to a log server) containing the command or exit code and all environment variables. By abstracting the collection mechanism from zsh, we can easily upgrade either component.The functions used are:

preexec If this function is defined, it is executed each time before the command line is run. This lets us know what the command is and when it is started by calling the helper program with the command line.

precmd If this function is defined, it is executed each time before the shell prompt is displayed. We use this to know when the command has ended, and to call the helper program with the exit code.

When a user's account is modified to use ZSH, they are told how to disable the logging with the command "`BigBrother off`." As well, users are informed when each shell starts that they are contributing to data collection, and reminded on every command line that "Big Brother is Watching" through another ZSH feature, the right hand prompt.

The helper program and example ZSH scripts are currently available from `http://www.cs.ualberta.ca/~benjamin/AUI/NotSoBigBrother-0.1.tar.gz`.

Davison and Hirsh collected data for a lesser period (between two and six months)[7, 6], but greater number of subjects (77) than we did. Similarly, all subjects knew their sessions were being observed and could turn it off. They also ran into problems with the data collection — they exploited the history mechanism in a manner similar to our initial data collection — and as a result, lost history traces when sessions terminated abnormally[1].

[1] We encountered similar situations with our initial data collection. Later data collection was done on a remote machine and thus avoided such pitfalls.